MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No. 958a March 1988

Obviously Synchronizable Series Expressions: Part I: User's Manual for the OSS Macro Package

by

Richard C. Waters

Abstract

The benefits of programming in a functional style are well known. In particular, algorithms that are expressed as compositions of functions operating on series/vectors/streams of data elements are much easier to understand and modify than equivalent algorithms expressed as loops. Unfortunately, many programmers hesitate to use series expressions, because they are typically implemented very inefficiently.

A Common Lisp macro package (OSS) has been implemented which supports a restricted class of series expressions, *obviously synchronizable series expressions*, which can be evaluated very efficiently by automatically converting them into loops. Using this macro package, programmers can obtain the advantages of expressing computations as series expressions without incurring any run-time overhead.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research has been provided in part by the National Science Foundation under grant IRI-8616644, in part by the IBM Corporation, in part by the NYNEX Corporation, and in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-85-K-0124.

The views and conclusions contained in this document are those of the authors, and should not be interpreted as representing the policies, neither expressed nor implied, of the National Science Foundation, of the IBM Corporation, of the NYNEX Corporation, or of the Department of Defense.

Contents

Acknowledgments. Both the OSS macro package and this report have benefited from the assistance of a number of people. In particular, C. Rich, A. Meyer, Y. Feldman, D. Chapman, and P. Anagnostopoulos made suggestions which led to a number of very significant improvements in the clarity and power of obviously synchronizable series expressions.

1. All You Need To Know to Get Started

This first section describes everything you need to know to start using the OSS macro package. It then presents a detailed example. Section 2 is a comprehensive reference manual. It describes the functions supported by the OSS macro package in detail. Section 3 contains the bibliography. Section 4 explains the warning and error messages that can be produced by the OSS macro package. Section 5 is both an index into Section 2 and an abbreviated description of the OSS functions.

A companion paper [6] gives an overview of the theory underlying the OSS macro package. It explains why things are designed the way they are and compares the OSS macro package with other systems that support operations on series. In addition, the companion paper gives a brief description of the algorithms used to implement the OSS macro package. As part of this, it describes a number of subprimitive constructs which are provided for advanced users of the OSS macro package.

The OSS data type. A *series* is an ordered linear sequence of elements. Vectors, lists, and streams are examples of series data types. The advantages (with respect to conciseness, understandability, and modifiability) of expressing algorithms as compositions of functions operating on series, rather than as loops, are well known. Unfortunately, as typically implemented, series expressions are very inefficient—so inefficient, that programmers are forced to use loops whenever efficiency matters.

Obviously Synchronizable Series (OSS) is a special series data type that can be implemented extremely efficiently by automatically converting OSS expressions into loops. This allows programmers to gain the benefit of using series expressions without paying any price in efficiency.

The OSS macro package adds support for the OSS data type to Common Lisp [4]. The macro package was originally developed under version 7 of the Symbolics Lisp Machine software [7]. However, it is written in standard Common Lisp and should be able to run in any implementation of Common Lisp. (It has been tested in DEC Common Lisp and Sun Common Lisp as well as Symbolics Common Lisp.)

The basic functionality provided by the OSS macro package is similar to the functionality provided by the Common Lisp sequence functions. However, in addition to being much more efficient, the OSS macro package is more powerful than the sequence functions, because it includes almost all of the operations supported by APL [3] and by the Loop macro [2]. As a result, OSS expressions go much farther than the sequence functions towards the goal of eliminating the need for explicit loops.

Predefined OSS functions. The heart of the OSS macro package is a set of several dozen functions which operate on OSS series. These functions divide naturally into three classes. *Enumerators* produce series without consuming any. *Transducers* compute series from series. *Reducers* consume series without producing any. As a mnemonic device, the name of each predefined OSS function begins with a letter code that indicates the type of operation. These letters are intended to be pronounced as separate syllables.

Predefined enumerators include `Elist` which enumerates successive elements of a list, `Evector` which enumerates the elements of a vector, and `Eup` which enumerates the integers in a range. (The notation `[...]` is used to represent an OSS series.)

```
(Elist '(a b c)) ⇒ [a b c]
(Evector '#(a b c)) ⇒ [a b c]
(Eup 1 :to 3) ⇒ [1 2 3]
```

Predefined transducers include `Tpositions` which returns the positions of the non-null elements in a series and `Tselect` which selects the elements of its second argument which correspond to non-null elements of its first argument.

```
(Tpositions [a nil b c nil nil]) ⇒ [0 2 3]
(Tselect [nil T T nil] [1 2 3 4]) ⇒ [2 3]
```

Predefined reducers include `Rlist` which combines the elements of a series into a list, `Rsum` which adds up the elements of a series, `Rlength` which computes the length of a series, and `Rfirst` which returns the first element of a series.

```
(Rlist [a b c]) ⇒ (a b c)
(Rsum [1 2 3]) ⇒ 6
(Rlength [a b c]) ⇒ 3
(Rfirst [a b c]) ⇒ a
```

As simple illustrations of how OSS functions are used, consider the following.

```
(Rsum (Evector '#(1 2 3))) ⇒ 6
(Rlist (Tpositions (Elist '(a nil b c nil)))) ⇒ (0 2 3)
```

Higher-Order OSS functions. The OSS macro package provides a number of higher-order functions which support general classes of OSS operations. (Each of these functions end in the suffix "F", which is pronounced separately.)

For example, enumeration is supported by (`EnumerateF` *init step test*). This enumerates an OSS series of elements starting with *init* by repeatedly applying *step*. The OSS series consists of the values up to, but not including, the first value for which *test* is true.

Reduction is supported by (`ReduceF` *init function items*) which is analogous to the sequence function `reduce`. The elements of the OSS series *items* are combined together using *function*. The quantity *init* is used as an initial seed value for the accumulation.

Mapping is supported by (`TmapF` *function items*) which is analogous to the sequence function `map`. A series is computed by applying *function* to each element of *items*.

```
(EnumerateF 3 #'1- #'minusp) ⇒ [3 2 1 0]
(ReduceF 0 #'+ [1 2 3]) ⇒ 6
(TmapF #'sqrt [4 9 16]) ⇒ [2 3 4]
```

Implicit mapping. The OSS macro package contains a special mechanism that makes mapping particularly easy. Whenever an ordinary Lisp function is applied to an OSS series, it is automatically mapped over the elements of the OSS series. For example, in the expression below, the function `sqrt` is mapped over the OSS series of numbers created by `Evector`.

```
(Rsum (sqrt (Evector '#(4 16))))
  ≡ (Rsum (TmapF #'sqrt (Evector '#(4 16)))) ⇒ 6
```

To a considerable extent, implicit mapping is a peripheral part of the OSS macro package—one can always use `TmapF` instead. However, due to the ubiquitous nature of mapping, implicit mapping is extremely convenient. As illustrated below, its key virtue is that it reduces the number of literal `lambda` expressions that have to be written.

```
(Rsum (expt (abs (Evector '#(2 -2 3))) 3))
  ≡ (Rsum (TmapF #'(lambda (x) (expt (abs x) 3))
                  (Evector '#(2 -2 3)))) ⇒ 43
```

Creating OSS variables. The OSS macro package provides two forms (`letS` and `letS*`) which are analogous to `let` and `let*`, except that they make it possible to create variables that can hold OSS series. (The suffix "S", pronounced separately, is used to indicate primitive OSS forms.) As shown in the example below, `letS` can be used to bind both ordinary variables (e.g., `n`) and OSS variables (e.g., `items`).

```
(defun average (v)
  (letS* ((items (Evector v))
          (sum (Rsum items))
          (n (Rlength items)))
    (/ sum n)))
(average '#(1 2 3)) ⇒ 2
```

User-defined OSS functions. New OSS functions can be defined by using the form `defunS` which is analogous to `defun`. Explicit declarations are required inside `defunS` to indicate which arguments receive OSS series. The following example shows the definition of an OSS function which computes the product of the numbers in an OSS series.

```
(defunS Rproduct (numbers)
    (declare (type oss numbers))
  (ReduceF 1 #'* numbers))
(Rproduct [2 4 6]) ⇒ 48
```

Restrictions on OSS expressions. As illustrated by the examples above, OSS expressions are constructed in the same way as any other Lisp expression—i.e., OSS functions are composed together in any way desired. However, in order to guarantee that OSS expressions can always be converted into highly efficient loops, a few restrictions have to be followed. These restrictions are summarized in the beginning of Section 2 and discussed in detail in [6].

Here, it is sufficient to note that the OSS macro package is designed so that it is impossible to violate most of the restrictions. The remaining restrictions are checked by the macro package and any violations are automatically fixed. However, warning messages are issued whenever a violation is detected, because, as discussed in the beginning of Section 2, it is often possible for the user to fix a violation in a way which is much more efficient than the automatic fix supplied by the macro package.

The best approach for programmers to take is to simply write OSS expressions without worrying about the restrictions. In this regard, it should be noted that simple OSS expressions are very unlikely to violate any of the restrictions. In particular, it is impossible for

an OSS expression to violate any of the restrictions unless it contains a variable bound by `letS` or `defunS`. When violations do occur, they can either be ignored (since they cannot lead to incorrect results) or dealt with on an individual basis (which is advisable since violations can lead to significant inefficiencies).

Benefits. The benefit of OSS expressions is that they retain most of the advantages of functional programming using series, while eliminating the costs. However, given the restrictions alluded to above, the question naturally arises as to whether OSS expressions are applicable in a wide enough range of situations to be of real pragmatic benefit.

An informal study [5] was undertaken of the kinds of loops programmers actually write. This study suggests that approximately 80% of the loops programmers write are constructed by combining a few common kinds of looping algorithms. The OSS macro package is designed so that all of these algorithms can be represented as OSS functions. As a result, it appears that approximately 80% of loops can be trivially rewritten as OSS expressions. Many more can be converted to this form with only minor modification.

Moreover, the benefits of using OSS expressions go beyond replacing individual loops. A major shift toward using OSS expressions would be a significant change in the way programming is done. At the current time, most programs contain one or more loops and most of the interesting computation in these programs occurs in these loops. This is quite unfortunate, since loops are generally acknowledged to be one of the hardest things to understand in any program. If OSS expressions were used whenever possible, most programs would not contain any loops. This would be a major step forward in conciseness, readability, verifiability, and maintainability.

Example

The following example shows what it is like to use OSS expressions in a realistic programming context. The example consists of two parts: a pair of functions which convert between sets represented as lists and sets represented as bits packed into an integer and a graph algorithm which uses the integer representation of sets.

Bit sets. Small sets can be represented very efficiently as binary integers where each 1 bit in the integer represents an element in the set. Below, sets represented in this fashion are referred to as *bit sets*.

Common Lisp provides a number of bitwise operations on integers which can be used to manipulate bit sets. In particular, `logior` computes the union of two bit sets while `logand` computes their intersection.

The functions in Figure 1.1 convert between sets represented as lists and bit sets. In order to perform this conversion, a mapping has to be established between bit positions and potential set elements. This mapping is specified by a *universe*. A universe is a list of elements. If a bit set b is associated with a universe u, then the ith element in u is in the set represented by b iff the ith bit in b is 1.

For example, given the universe `(a b c d e)`, the integer `#b01011` represents the set `{a,b,d}`. (By Common Lisp convention, the 0th bit in an integer is the rightmost bit.)

Given a bit set and its associated universe, the function `bset->list` converts the bit set into a set represented as a list of its elements. It does this by enumerating the elements in the universe along with their positions and constructing a list of the elements

```
(defun bset->list (bset universe)
  (Rlist (Tselect (logbitp (Eup 0) bset) (Elist universe))))

(defun list->bset (list universe)
  (ReduceF 0 #'logior (ash 1 (bit-position (Elist list) universe))))

(defun bit-position (item universe)
  (or (Rfirst (Tpositions (eq item (Elist universe))))
      (1- (length (nconc universe (list item))))))
```

Figure 1.1: Converting between lists and bit sets.

which correspond to 1s in the integer representing the bit set. (When no `:to` argument is supplied, `Eup` counts up forever.)

The function `list->bset` converts a set represented as a list of its elements into a bit set. Its second argument is the universe which is to be associated with the bit set created. For each element of the list, the function `bit-position` is called in order to determine which bit position should be set to 1. The function `ash` is used to create an integer with the correct bit set to 1. The function `ReduceF` is used to combine the integers corresponding to the individual elements together into a bit set corresponding to the list.

The function `bit-position` takes an item and a universe and returns the bit position corresponding to the item. The function operates in one of two ways depending on whether or not the item is in the universe. The first line of the function contains an OSS expression which determines the position of the item in the universe. If the item is not in the universe, the expression returns `nil`. (The function `Rfirst` returns `nil` if it is passed a series of length zero.)

If the item is not in the universe, the second line of the function adds the item onto the end of the universe and returns its position. The extension of the universe is done be side-effect so that it will be permanently recorded in the universe.

Figure 1.2 shows the definition of two OSS reducers which operate on OSS series of bit sets. The first function computes the union of a series of bit sets, while the second computes their intersection.

Live variable analysis. As an illustration of the way bit sets might be used, consider the following. Suppose that in a compiler, program code is being represented as blocks of straight-line code connected by possibly cyclic control flow. The top part of Figure 1.3 shows the data structure which represents a block of code. Each block has several pieces of information associated with it. Two of these pieces of information are the blocks

```
(defunS Rlogior (bsets)
    (declare (type oss bsets))
  (ReduceF 0 #'logior bsets))

(defunS Rlogand (bsets)
    (declare (type oss bsets))
  (ReduceF -1 #'logand bsets))
```

Figure 1.2: Operations on OSS series of bit sets.

that can branch to the block in question and the blocks it can branch to. A program is represented as a list of blocks that point to each other through these fields.

In addition to control flow information, each structure contains information about the way variables are accessed. In particular, it records the variables that are written by the block and the variables that are used by the block (i.e., either read without being written or read before they are written). An additional field (computed by the function `determine-live` discussed below) records the variables which are *live* at the end of the block. (A variable is live if it has to be saved, because it can potentially be used by a following block.) Finally, there is a temporary data field which is used by functions (such as `determine-live`) which perform computations involved with the blocks.

The remainder of Figure 1.3 shows the function `determine-live` which, given a program represented as a list of blocks, determines the variables which are live in each block. To perform this computation efficiently, the function uses bit sets. The function operates in three steps. The first step (`convert-to-bsets`) looks at each block and sets up an auxiliary data structure containing bit set representations for the written variables, the used variables, and an initial guess that there are no live variables. This auxiliary structure is defined by the third form in Figure 1.3 and is stored in the `temp` field of the block. The integer 0 represents an empty bit set.

The second step (`perform-relaxation`) determines which variables are live. This is done by relaxation. The initial guess that there are no live variables in any block is successively improved until the correct answer is obtained.

The third step (`convert-from-bsets`) operates in the reverse of the first step. Each block is inspected and the bit set representation of the live variables is converted into a list which is stored in the live field of the block.

On each cycle of the loop in `perform-relaxation`, a block is examined to determine whether its live set has to be changed. To do this (see the function `live-estimate`), the successors of the block are inspected. Each successor needs to have available to it the variables it uses, plus the variables that are supposed to be live after it, minus the variables it writes. (The function `logandc2` takes the difference of two bit sets.) A new estimate of the total set of variables needed by the successors as a group is computed by using `Rlogior`.

If this new estimate is different from the current estimate of what variables are live, then the estimate is changed. In addition, if the estimate is changed, `perform-relaxation` has to make sure that all of the predecessors of the current block will be examined to see if the new estimate for the current block requires their live estimates to be changed. This is done by adding each predecessor onto the list `to-do` unless it is already there. As soon as the estimates of liveness stop changing, the computation can stop.

Summary. The function `determine-live` is a particularly good example of the way OSS expressions are intended to be used in two ways. First, OSS expressions are used in a number of places to express computations which would otherwise be expressed less clearly as loops or less efficiently as sequence function expressions. Second, the main relaxation algorithm is expressed as a loop. This is done, because neither OSS expressions (nor Common Lisp sequence function expressions) lend themselves to expressing the relaxation algorithm. This highlights the fact that OSS expressions are not intended to render loops entirely obsolete, but rather to provide a greatly improved method for expressing the

vast majority of loops.

```
(defstruct (block (:conc-name nil))
  predecessors ;Blocks that can branch to this one.
  successors   ;Blocks this one can branch to.
  written      ;Variables written in the block.
  used         ;Variables read before written in the block.
  live         ;Variables that must be available at exit.
  temp)        ;Temporary storage location.

(defun determine-live (program-graph)
  (let ((universe (list nil)))
    (convert-to-bsets program-graph universe)
    (perform-relaxation program-graph)
    (convert-from-bsets program-graph universe))
  program-graph)

(defstruct (temp-bsets (:conc-name bset-))
  used written live)

(defun convert-to-bsets (program-graph universe)
  (letS ((block (Elist program-graph)))
    (setf (temp block)
          (make-temp-bsets
            :used (list->bset (used block) universe)
            :written (list->bset (written block) universe)
            :live 0))))

(defun perform-relaxation (program-graph)
  (let ((to-do program-graph))
    (loop
      (when (null to-do) (return (values)))
      (let* ((block (pop to-do))
             (estimate (live-estimate block)))
        (when (not (= estimate (bset-live (temp block))))
          (setf (bset-live (temp block)) estimate)
          (letS ((prev (Elist (predecessors block))))
            (pushnew prev to-do)))))))

(defun live-estimate (block)
  (letS ((next (temp (Elist (successors block)))))
    (Rlogior (logior (bset-used next)
                     (logandc2 (bset-live next)
                               (bset-written next))))))

(defun convert-from-bsets (program-graph universe)
  (letS ((block (Elist program-graph)))
    (setf (live block)
          (bset->list (bset-live (temp block)) universe))
    (setf (temp block) nil)))
```

Figure 1.3: Live variable analysis.

2. Reference Manual

This section is organized around descriptions of the various functions and forms supported by the OSS macro package. Each description begins with a header which shows the arguments and results of the function or form being described. For ease of reference, the headers are duplicated in Section 5. In Section 5, the headers are in alphabetical order and show the page where the function or form is described.

In a reference manual like this one, it is advantageous to describe each construct separately and completely. However, this inevitably leads to presentation problems, because everything is related to everything else. Therefore, one cannot avoid referring to things which have not been discussed. The reader is encouraged to skip around in the document and to realize that more than one reading will probably be necessary in order to gain a complete understanding of the OSS macro package.

Although the following list of OSS functions is large, it should not be taken as complete. Every effort has been made to provide a wide range of useful predefined functions. However, except for a few primitive forms, all of these functions could have been defined by the user. It is hoped that users will write many more such functions. A key reason for presenting a wide array of predefined functions is to inspire users with thoughts of the wide variety of functions they can write for themselves.

Restrictions and Definitions of Terms.

As alluded to in Section 1, there are a number of restrictions which OSS expressions have to obey. The OSS macro package is designed so that all but three of these restrictions are impossible to violate with the facilities provided. As a result, the programmer need not think about these restrictions at all.

The OSS macro package checks to see that the remaining three restrictions are obeyed on an expression by expression basis and automatically fixes any violations which are detected. However, the automatic fixes are often not very efficient. As a result, it is advisable for the user to fix such violations explicitly.

Given that simple OSS expressions are very unlikely to violate any of the restrictions, and any violations which do occur are automatically fixed, it is reasonable for the reader to skip this section when first reading this manual. However, it is useful to read this section before trying to write complex OSS expressions.

The discussion below starts by defining two key terms (on-line functions and early termination) which are used to categorize the OSS functions described in the rest of this manual. The discussion then continues by briefly describing the three restrictions which can be violated. (See [6] for a complete discussion of all the restrictions.)

On-line and off-line. Suppose that f is an OSS function which reads one or more series inputs and writes one or more series outputs. The function f is *on-line* [1] if it operates in the following fashion. First, f reads in the first element of each input series, then it writes out the first element of each output series, then it reads in the second element of each input series, then it writes out the second element of each output series, and so on. In addition, f must immediately terminate as soon as any input runs out of

elements. If a f is not on-line, then it is *off-line*.

In the context of OSS expressions, the term on-line is generalized so that it applies to individual OSS input and output ports in addition to whole functions. An OSS port is on-line iff the processing at that port always follows the rigidly synchronized pattern described above. Otherwise, it is off-line. From this point of view, a function is on-line iff all of its OSS ports are on-line.

The prototypical example of an on-line OSS function is `TmapF` (which maps a function over a series). Each time it reads an input element it applies the mapped function to it and writes an output element. In contrast, the function `Tremove-duplicates` (which removes the duplicate elements from a series) is not on-line. Since some of the input elements do not become output elements, it is not possible for `Tremove-duplicates` to write an output element every time it reads an input element.

For every OSS function, the documentation below specifies which ports are on-line and which are off-line. In this regard, it is interesting to note that every function which has only one OSS port (e.g., enumerators with only one output and reducers with only one input) are trivially on-line. The only OSS functions which have off-line ports are transducers.

Early termination. An important feature of OSS functions is the situations under which they terminate. The definition of on-line above requires that on-line functions must terminate as soon as any series input runs out of elements. If an OSS function can terminate before any of its inputs are exhausted, then it is an *early terminator*. The degenerate case of functions which do not have any series inputs (i.e., enumerators) is categorized by saying that enumerators are early terminators iff they can terminate.

As an example of an early terminator, consider the function `TuntilF` (which reads a series and returns all of the elements of that series up to, but not including, the first element which satisfies a given predicate). This function is an early terminator, because it can terminate before the input runs out of elements.

The documentation below specifies which functions are early terminators. Besides enumerators, their are only 7 OSS functions which are early terminators.

Isolation. A data flow arc δ in an OSS expression X is isolated iff it is possible to partition the functions in X into two parts Y and $\overline{Y}$ in such a way that: δ goes from Y to $\overline{Y}$, there is no OSS data flow from Y to $\overline{Y}$, and there is no data flow from $\overline{Y}$ to Y. For example, consider the OSS expression `(letS ((x (f y))) (i (h x (g x))))` which corresponds to the graph in Figure 2.1.

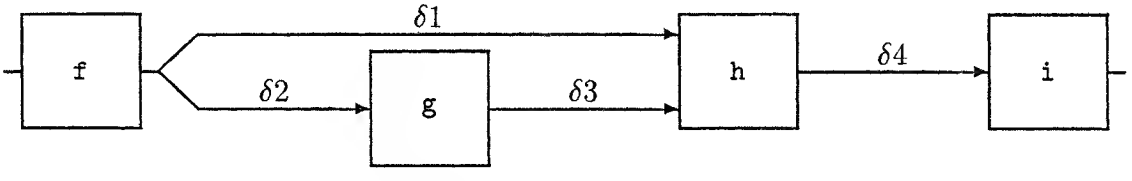

Figure 2.1: Parallel data flow paths in an expression.

The data flow arc $\delta 4$ is isolated. To show this, one merely has to partition the expression so that `f`, `g`, and `h` are on one side and `i` is on the other. The question of

whether or not the other data flow arcs are isolated is more complicated to answer. If $\delta 3$ crosses a partition, then $\delta 1$ must cross this partition as well. As a result, $\delta 3$ is isolated iff $\delta 1$ carries a non-OSS value. (This is true no matter what kind of value passes over $\delta 3$ itself.) In a related situation, $\delta 2$ is isolated iff (it and therefore $\delta 1$) carries a non-OSS value. Finally, consider the arc $\delta 1$. Here there are two potential partitions to consider: one which cuts $\delta 2$ and one which cuts $\delta 3$. The data flow arc $\delta 1$ is isolated iff either it (and therefore $\delta 2$) or $\delta 3$ carries a non-OSS value.

The concept of isolation is extended to inputs and outputs as follows. An output p in an expression X is isolated iff X can be partitioned into two parts Y and $\overline{Y}$ such that: every data flow originating on p goes from Y to $\overline{Y}$, every other data flow from Y to $\overline{Y}$ is a non-OSS data flow, and there is no data flow from $\overline{Y}$ to Y. An input q in an expression X is isolated iff X can be partitioned into two parts Y and $\overline{Y}$ such that: the data flow terminating on q goes from Y to $\overline{Y}$, every other data flow from Y to $\overline{Y}$ is a non-OSS data flow, and there is no data flow from $\overline{Y}$ to Y.

For example, in Figure 2.1, the outputs of `f` and `h` are isolated as is the input of `i`. The input and output of `g` are isolated iff `f` computes a non-OSS value. The inputs of `h` are isolated iff the data flow arcs terminating on them are isolated.

Non-OSS data flows must be isolated. In order for an OSS expression to be reliably converted into a highly efficient loop, every non-OSS data flow in it must be isolated. As an example of an expression where this is not true, consider the following. In this expression, the data flow implemented by the variable `total` is not isolated.

```
(letS* ((nums (Evector '#(3 2 8)))              ;Signals warning 16
        (total (ReduceF 0 #'+ nums)))
  (Rvector (/ nums total))) ⇒ #(3/13 2/13 8/13)
```

(The basic problem here is that while the elements created by `Evector` are being used to compute `total`, they all have to be saved so that they can be used again later in order to perform the indicated divisions. Eliminating the need for such storage is the key source of efficiency underlying OSS expressions.)

Off-line OSS ports must be isolated. In order for an OSS expression to be reliably converted into a highly efficient loop, every off-line port must be isolated. As an example of an expression which has an off-line output which is not isolated, consider the following. In this expression, the data flow implemented by the variable `positions` is not isolated.

```
(letS* ((keys (Elist list))                     ;Signals warning 17.1
        (positions (Tpositions keys)))
  (Rlist (list positions keys)))
```

(The basic problem here is that since `Tpositions` skips null elements of the input, `Tpositions` sometimes has to read several input elements before it can produce the next output element. This forces an unpredictable number of elements of `keys` to be saved so that they can be used later when creating lists. As above, eliminating the need for such storage is the main goal of OSS expressions.)

Code copying. If an OSS expression violates either of the above restrictions, the OSS macro packaged automatically fixes the problem by copying code until the data flow or port in question becomes isolated. For instance, the example above of an OSS expression in which a non-OSS data flow is not isolated is fixed as follows.

```
(letS* ((nums (Evector '#(3 2 8)))
        (total (ReduceF 0 #'+ (Evector '#(3 2 8)))))
  (Rvector (/ nums total))) ⇒ #(3/13 2/13 8/13)
```

Even though the problem has been automatically fixed, the OSS macro package issues a warning message. This is done for two reasons. First, if side-effects (e.g., input or output) are involved, the code copying that was performed may not be correctness preserving. Second, large amounts of code sometimes have to be copied and that can introduce large amounts of extra computation.

A major goal of OSS expressions is ensuring that expressions which *look* simple to compute actually are simple to compute. Automatically introducing large amounts of additional computation without the programmer's knowledge would violate this goal. At the very least, issuing warning messages makes programmers aware of what is expensive to compute and what is not. Looked at from a more positive perspective, it encourages them to think of ways to compute what they want without code copying being required.

For instance, consider the example above of an OSS expression in which an off-line port is not isolated. It might be the case that the programmer knows that `list` does not contain any null elements and that `Tpositions` is therefore merely being used to enumerate what the positions of the elements are. In this situation, the expression can be fixed as follows, which does not require any code copying. (The key insight here is that the positions do not actually depend on the values in the list.)

```
(let ((list '(a b c)))
  (letS* ((keys (Elist list))
          (positions (Eup 0)))
    (Rlist (list positions keys)))) ⇒ ((0 a) (1 b) (2 c))
```

(It is interesting to note that if an expression is a tree (as opposed to a graph as in Figure 2.1), then every data flow arc and every port is isolated. This is why OSS expressions which do not contain variables bound by `letS`, `lambdaS`, or `defunS` cannot violated either of the isolation restrictions. This is also why code copying can always fix any violation—code copying can convert any graph into a tree.)

On-line subexpressions. The two isolation restrictions above permit a divide and conquer approach to the processing of OSS expressions. If an OSS expression obeys the isolation restrictions, then it can be repeatedly partitioned until all of the data flow in each subexpression goes from an on-line output to an on-line input. The subexpressions which remain after partitioning are referred to as *on-line subexpressions.*

Termination points. The functions in each on-line subexpression can be divided into two classes: those which are termination points and those which are not. A function is a termination point if it can terminate before any other function in the subexpression terminates. There are two reasons for functions being termination points. Functions which are early terminators are always termination points. In addition, any function which reads an OSS series which comes from a different on-line subexpression is a termination point.

Data flow paths between termination points and outputs. In order for an OSS expression to be reliably converted into a highly efficient loop, it must be the case

that within each on-line subexpression, there is a data flow path from each termination point to each output. As an example of an OSS expression for which this property does not hold, consider the following. Partitioning divides this expression into two on-line subexpressions, one containing `list` and one containing everything else. In the large on-line subexpression, the two instances of `Evector` are termination points. The program violates the property above, because there is no data flow path from the termination point `(Evector weight-vector)` to the output of `(Rvector squares)`.

```
(defun weighted-squares (value-vector weight-vector)
  (letS* ((values (Evector value-vector))          ;Signals warning 18
          (weights (Evector weight-vector))
          (squares (* values values))
          (weighted-squares (* squares weights)))
    (list (Rvector squares) (Rvector weighted-squares))))

(weighted-squares #(1 2 3) #(2 3 4)) ⇒ (#(1 4 9) #(2 12 36))
(weighted-squares #(1 2) #(2 3 4)) ⇒ (#(1 4) #(2 12))
(weighted-squares #(1 2 3) #(2 3)) ⇒ (#(1 4 9) #(2 12))
```

(The basic problem here is that if the number of elements in `value-vector` is greater than the number of elements in `weight-vector`, the computation of `squares` has to continue even after the computation of `weighted-squares` has been completed. This kind of partial continuing evaluation in a single on-line subexpression is not supported by the OSS macro package, because it was judged that it requires too much overhead in order to control what gets evaluated when.)

When an OSS expression violates the restriction above, the violation is automatically fixed by applying code copying. It is impossible for an on-line subexpression to violate the restriction unless it computes two different outputs. Code copying can always be used to break the subexpression in question into two parts each of which computes one of the outputs. Unfortunately, this can require a great deal of code to be copied. There are two basic approaches which can be used to fix a violation much more efficiently: reducing the number of termination points and increasing the connectivity between termination points and outputs.

The easiest way to decrease the number of termination points is to replace early terminators by equivalent operations which are not early terminators (for example, see page 37). If an early terminator is not an enumerator, then this can always be done without difficultly. (The documentation below describes a non-early variant for each early terminating transducer and reducer.) If multiple enumerators are the problem (as in the example above) decreasing the number of termination points is usually not practical. However, sometimes an enumerator which terminates can be replaced by an enumerator which never terminates.

The connectivity between termination points and outputs can be increased by using the function `Tcotruncate`. As discussed on page 26, this is the preferred way to fix the problem in the example above.

General Information

Before discussing the individual OSS functions in detail, a few general comments are in order. First, all of the OSS functions and forms are defined in the package `OSS`. To make

these names easily accessible, *use* the package `OSS` (i.e., evaluate `(use-package "OSS")`). If this is not done, the names will have to be prefixed with "`oss:`" when they are used.

Naming conventions. The names of the various OSS functions and forms follow a strict naming convention. The first letter of an OSS function name indicates the type of function as shown below. The letter codes are written in upper case in this document (case does not matter to Common Lisp) and each letter is intended to be pronounced as a separate syllable.

`E` Enumerator.
`T` Transducer.
`R` Reducer.

The last letter of each OSS special form is "`S`". In general, this indicates that the form is primitive in the sense that it could not be defined by the user. Some OSS functions end in the letter "`F`". This is used to indicate that the function is a higher-order function which takes functions as arguments.

The naming convention has two advantages: one trivial but vital and the other more fundamentally useful. First, many of the OSS functions are very similar to standard Common Lisp sequence functions. As a result, it makes sense to give them similar names. However, it is not possible to give them exactly the same names without redefining the standard functions. The naming convention allows the names to be closely related in a predictable way without making the names unreasonably long.

Second, the naming convention highlights several properties of OSS functions which make it easier to read and understand OSS expressions. In particular, the prefixes highlight the places where series are created and consumed.

The names of arguments and results of OSS functions are also chosen following naming conventions. First, all of the names are chosen in an attempt to indicate type restrictions (e.g., *number* indicates that an argument must be a number; *item* indicates that there is no type restriction). Plural names are used iff the value in question is an OSS series (e.g., *numbers* indicates an OSS series of numbers; *items* indicates an OSS series of unrestricted values). The name of a series input or output begins with "`O`" iff it is off-line.

OSS series. Two general points about OSS series are worthy of note. First, like Common Lisp sequences, OSS series use zero-based indexing (i.e., the first element is the 0th element). Second, unlike Common Lisp sequences, OSS series can be unbounded in length.

Tutorial mode. A prominent feature of the various descriptions is that they contain many examples. These examples contain large numbers of OSS series as inputs and outputs. In the interest of brevity, the notation `[...]` is used to indicate a literal OSS series. If the last entry in a literal OSS series is an ellipsis, this indicates that the OSS series is unbounded in length.

```
[1 2 3]
[a b (c d)]
[T nil T nil ...]
```

The notation `[...]` is not supported by the OSS macro package. It would be straightforward to do so by using `set-macro-character`. Perhaps even better, one could use

`set-dispatch-macro-character` to support a notation `#[...]` analogous to `#(...)`. However, although literal series are very useful in the examples below, experience suggests that literal series are seldom useful when writing actual programs. Inasmuch as this is the case, it was decided that it was unwise to use up one of the small set of characters which are available for user-defined reader macros or user-defined # dispatch characters.

Many of the examples show OSS expressions returning OSS series as their values. However, one should not take this literally. If these examples are typed to Common Lisp as isolated expressions, they will not return any values. This is so, because the OSS macro package does not allow complete OSS expressions to return OSS series. The examples are intended to show what would be returned if the example expressions were nested in larger expressions.

- `oss-tutorial-mode &optional` (*T-or-nil* `T`) ⇒ *state-of-tutorial-mode*

The above not withstanding, the OSS macro package provides a special tutorial mode in which the notation `[...]` is supported and OSS expressions can return (potentially unbounded) OSS values. However, these values still cannot be stored in ordinary variables. This mode is entered by calling the function `oss-tutorial-mode` with an argument of `T`. Calling the function with an argument of `nil` turns tutorial mode off.

Using tutorial mode, it is possible to directly duplicate the examples shown below. However, tutorial mode is very inefficient. What is worse, tutorial mode introduces *non-correctness-preserving* changes in OSS expressions. (For example, in order to correctly duplicate the examples that illustrate error messages about non-terminating expressions and the fact that OSS series are not actually returned by complete OSS expressions, tutorial mode must be turned off.) All in all, it is important that tutorial mode not be used as anything other than an educational aid.

OSS functions are actually macros. Every OSS function is actually a macro. As a result, OSS functions cannot be `funcall`'ed, or `apply`'ed. When the user defines new OSS functions, they must be defined before the first time they are used. Also, when an OSS function takes keyword arguments, the keywords must be literals. They cannot be expressions which evaluate to keywords at run time.

Finally, the macro expansion processing associated with OSS expressions is relatively time consuming. In order to avoid this overhead during the running of a user program, it is important that programs containing OSS expressions be compiled rather than run interpretively.

A minor advantage of the fact that everything in the OSS macro package is a macro is that once a program which uses the macro package is compiled, the compiled program can subsequently be run without having to load the OSS macro package.

A more important advantage of the fact that everything in the OSS macro package is a macro is that quoted macro names can be used as functional arguments to higher-order OSS functions. (In contrast, quoted macro names cannot be used as functional arguments to higher-order Common Lisp functions such as `reduce`.) Although this may appear to be a minor benefit, it is actually quite useful.

Enumerators

Enumerators create OSS outputs based on non-OSS inputs. There are two basic kinds of enumerators: ones that create an OSS series based on some formula (e.g., enumerating a sequence of integers) and ones that create an OSS series containing the elements of an aggregate data structure (e.g., enumerating the elements of a list). All the predefined enumerators are on-line. In general, they are all early terminators. However, as noted below, in some situations, some enumerators produce unbounded outputs and are not early terminators.

- `Eoss &rest` *expr-list* ⇒ *items*

  The *expr-list* consists of zero or more expressions. The function `Eoss` creates an OSS series containing the values of these expressions. Every expression in *expr-list* is evaluated before the first output element is returned.

```
(Eoss 1 'a 'b) ⇒ [1 a b]
(Eoss) ⇒ []
```

  To get the effect of delaying the evaluation of individual elements until they are needed, it is necessary to define a special purpose enumerator which computes the individual items as needed. However, due to the control overhead required, this is seldom worthwhile.

  It is possible for the *expr-list* to contain an instance of `:R`. (This must be a literal instance of `:R`, not an expression which evaluates to `:R`.) If this is the case, then `Eoss` produces an unbounded OSS series analogous to a repeating decimal number. The output consists of the values of the expressions preceding the `:R` followed by an unbounded number of repetitions of the values following the `:R`, if there are any such values. (In this situation, `Eoss` is not an early terminator.)

```
(Eoss 1 'a :R 'b 'c) ⇒ [1 a b c b c b c ...]
(Eoss T :R nil) ⇒ [T nil nil nil ...]
(Eoss 1 :R) ⇒ [1]
(Eoss :R 1) ⇒ [1 1 1 ...]
```

- `Eup &optional (`*start* `0) &key (:by 1) :to :below :length` ⇒ *numbers*

  This function is analogous to the Loop macro [2] numeric iteration clause. It creates an OSS series of numbers starting with *start* and counting up by `:by`. The argument *start* is optional and defaults to integer `0`. The keyword argument `:by` must always be a positive number and defaults to integer `1`.

  There are four kinds of end tests. If `:to` is specified, stepping stops at this number. The number `:to` will be included in the OSS series iff `(- :to` *start*`)` is a multiple of `:by`. If `:below` is specified, things operate exactly as if `:to` were specified except that the number `:below` is never included in the OSS series. If `:length` is specified, the OSS series has length `:length`. It must be the case that `:length` is a non-negative integer. If `:length` is positive, the last element of the OSS series will be `(+ start (* :by (1- :length)))`. If none of the termination arguments are specified, the output has unbounded length. (In this situation, `Eup` is not an early terminator.) If more than one termination argument is specified, it is an error.

```
(Eup :to 4) ⇒ [0 1 2 3 4]
(Eup :to 4 :by 3) ⇒ [0 3]
(Eup 1 :below 4) ⇒ [1 2 3]
(Eup 4 :length 3) ⇒ [4 5 6]
(Eup) ⇒ [0 1 2 3 4 ...]
```

As shown in the following example, `Eup` does not assume that the numbers being enumerated are integers.

```
(Eup 1.5 :by .1 :length 3) ⇒ [1.5 1.6 1.7]
```

- `Edown &optional (`*start* `0) &key (:by 1) :to :above :length` ⇒ *numbers*

The function `Edown` is analogous to `Eup`, except that it counts down instead of up and uses the keyword `:above` instead of `:below`.

```
(Edown :to -4) ⇒ [0 -1 -2 -3 -4]
(Edown :to -4 :by 3) ⇒ [0 -3]
(Edown 1 :above -4) ⇒ [1 0 -1 -2 -3]
(Edown 4 :length 3) ⇒ [4 3 2]
(Edown) ⇒ [0 -1 -2 -3 -4 ...]
(Edown -1.5 :by .1 :length 3) ⇒ [-1.5 -1.6 -1.7]
```

- `Esublists` *list* `&optional (`*end-test* `#'endp)` ⇒ *sublists*

This function creates an OSS series containing the successive sublists of *list*. The *end-test* must be a function from objects to boolean values (i.e., to null/non-null). It is used to determine when to stop the enumeration. Successive cdrs are returned up to, but not including, the first one for which *end-test* returns non-null.

```
(Esublists '(a b c)) ⇒ [(a b c) (b c) (c)]
(Esublists '(a b . c) #'atom) ⇒ [(a b . c) (b . c)]
```

The default *end-test* (`#'endp`) will cause `Esublists` to blow up if *list* contains a non-list cdr. More robust enumeration can be obtained by using the end-test `#'atom` as in the second example above. The assumption that *list* will end with `nil` is used as the default case, because the assumption sometimes allows programming errors to be detected closer to their sources.

- `Elist` *list* `&optional (`*end-test* `#'endp)` ⇒ *elements*

This function creates an OSS series containing the successive elements of *list*. It is closely analogous to `Esublists` as shown below. In particular, *end-test* has the same meaning and the same caveats apply.

```
(Elist '(a b c)) ⇒ [a b c]
(Elist '()) ⇒ []
(Elist '(a b . c) #'atom) ⇒ [a b]
(Elist list) ≡ (car (Esublists list))
```

The value returned by `Elist` can be used as a destination for `alterS`.

```
(let ((list '(a b c)))
  (alterS (Elist (cdr list)) (Eup))
  list) ⇒ (a 0 1)
```

- `Ealist` *alist* `&optional` (*test* `#'eql`) ⇒ *keys values*

  This function returns two OSS series containing keys and their associated values. The first element of *keys* is the key in the first entry in *alist*, the first element of *values* is the value in the first entry, and so on. The *alist* must be a proper list ending in `nil` and each entry in *alist* must be a cons cell or `nil`. Like `assoc`, `Ealist` skips entries which are `nil` and entries which have the same key as an earlier entry. The *test* argument is used to determine when two keys are the same.

```
(Ealist '((a . 1) () (a . 3) (b . 2))) ⇒ [a b] [1 2]
(Ealist nil) ⇒ [] []
```

  Both of the series returned by `Ealist` can be used as destinations for `alterS`. (In analogy with `multiple-value-bind`, `letS` can be used to bind both values returned by `Ealist`.)

```
(let ((alist '((a . 1) (b . 2))))
  (letS (((key val) (Ealist alist)))
    (alterS key (list key))
    (alterS val (1+ val)))
  alist) ⇒ '(((a) . 2) ((b) . 3))
```

  The OSS function `Ealist` is forced to perform a significant amount of computation in order to check that no duplicate keys or null entries are being enumerated. In a situation where it is known that no duplicate keys or null entries exist, it is much more efficient to use `Elist` as shown below.

```
(letS* ((e (Elist '((a . 1) (b . 2))))
        (keys (car e))
        (values (cdr e)))
  (Rlist (list keys values))) ⇒ ((a 1) (b 2))
```

- `Eplist` *plist* ⇒ *indicators values*

  This function returns two OSS series containing indicators and their associated values. The first element of *indicators* is the first indicator in the *plist*, the first element of *values* is the associated value, and so on. The *plist* argument must be a proper list of even length ending in `nil`. In analogy with the way `get` works, if an indicator appears more than once in *plist*, it (and its value) will only be enumerated the first time it appears. (Both of the OSS series returned by `Eplist` can be used as destinations for `alterS`.)

```
(Eplist '(a 1 a 3 b 2)) ⇒ [a b] [1 2]
(Eplist nil) ⇒ [] []
```

  The OSS function `Eplist` has to perform a significant amount of computation in order to check that no duplicate indicators are being enumerated. In a situation where it is known that no duplicate indicators exist, it is much more efficient to use `EnumerateF` as shown below.

```
(letS* ((e (EnumerateF '(a 1 b 2) #'cddr #'null))
        (indicators (car e))
        (values (cadr e)))
  (Rlist (list indicators values))) ⇒ ((a b) (1 2))
```

- `Etree` *tree* `&optional` (*leaf-test* `#'atom`) ⇒ *nodes*

This function creates an OSS series containing all of the nodes in *tree*. The function assumes that *tree* is a tree built of lists, where each node is a list and the elements in the list are the children of the node. The function `Etree` does not assume that the node lists end in `nil`; however, it ignores any non-list cdrs. (This behavior increases the utility of `Etree` when it is used to scan Lisp code.) The nodes in the tree are enumerated in preorder (i.e., first the root is output, then the nodes in the tree which is the first child of the root is enumerated in full, then the nodes in the tree which is the second child of the root is enumerated in full, etc.).

The *leaf-test* is used to decide which elements of the tree are leaves as opposed to internal nodes. Failure of the test should guarantee that the element is a list. By default, *leaf-test* is `#'atom`. This choice of test categorizes `nil` as a leaf rather than as a node with no children.

The function `Etree` assumes that *tree* is a tree as opposed to a graph. If *tree* is a graph instead of a tree (i.e. some node has more than one parent), then this node (and its descendants) will be enumerated more than once. If the tree is a cyclic graph, then the output series will be unbounded in length.

```
(Etree 'd) ⇒ [d]
(Etree '((c) d)) ⇒ [((c) d) (c) c d]
(Etree '((c) d)
       #'(lambda (e)
           (or (atom e) (atom (car e))))) ⇒ [((c) d) (c) d]
```

- `Efringe` *tree* `&optional` (*leaf-test* `#'atom`) ⇒ *leaves*

This enumerator is the same as `Etree` except that it only enumerates the leaves of the tree, skipping all internal nodes. The logical relationship between `Efringe` and `Etree` is shown in the first example below. However, `Efringe` is implemented more efficiently than this example would indicate.

```
(Efringe tree) ≡ (TselectF #'atom (Etree tree))
(Efringe 'd) ⇒ [d]
(Efringe '((c) d)) ⇒ [c d]
(Efringe '((c) d)
         #'(lambda (e)
             (or (atom e) (atom (car e))))) ⇒ [(c) d]
```

The value returned by `Efringe` can be used as a destination for `alterS`. However, if the entire tree is a leaf and gets altered, this will have no side-effect on the tree as a whole. In addition, altering a leaf will have no effect on the leaves enumerated. In particular, if a leaf is altered into a subtree, the leaves of this subtree will not get enumerated.

```
(let ((tree '((3) 4)))
  (letS ((leaf (Efringe tree)))
    (if (evenp leaf) (alterS leaf (- leaf))))
  tree) ⇒ ((3) -4)
```

- `Evector` *vector* `&optional` (*indices* `(Eup)`) ⇒ *elements*

  This function creates an OSS series of the elements of a one-dimensional array. If *indices* assumes its default value, `Evector` enumerates all of the elements of *vector* in order.

```
(Evector "BAR") ⇒ [#\B #\A #\R]
(Evector "") ⇒ []
```

  Looked at in greater detail, `Evector` enumerates the elements of *vector* which are indicated by the elements of the OSS series *indices*. The *indices* must be non-negative integers, however, they do not have to be in order. Enumeration stops when *indices* runs out, or an index greater than or equal to the length of `vector` is encountered. One can use `Eup` to create an index series which picks out a section of *vector*. (Since `Evector` takes in an OSS series it is technically a transducer, however, it is on-line and is an enumerator in spirit.)

```
(Evector '#(b a r) (Eup 1 :to 2)) ⇒ [a r]
(Evector "BAR" [0 2 1 1 4 1]) ⇒ [#\B #\R #\A #\A]
```

  The value returned by `Evector` can be used as a destination for `alterS`.

```
(let ((v "FOOBAR"))
  (alterS (Evector v (Eup 2 :to 4)) #\- v) ⇒ "FO---R"
```

- `Esequence` *sequence* `&optional` (*indices* `(Eup)`) ⇒ *elements*

  The function `Esequence` is the same as `Evector` except that it will work on any Common Lisp sequence. However, since it has to determine the type of *sequence* at run-time, it is much less efficient than either `Elist` or `Evector`. (The value returned by `Esequence` can be used as a destination for `alterS`.)

```
(Esequence '(b a r)) ⇒ [b a r]
(Esequence '#(b a r)) ⇒ [b a r]
```

- `Ehash` *table* ⇒ *keys values*

  This function returns two OSS series containing keys and their associated values. The first element of *keys* is the key of the first entry, the first element of *values* is the value in the first entry, and so on. (There are no guarantees as to the order in which entries will be enumerated.)

```
(Ehash (let ((h (make-hash-table)))
         (setf (gethash 'color h) 'brown)
         (setf (gethash 'name h) 'fred)
         h)) ⇒ [color name] [brown fred] ;in some order
```

  In the pure Common Lisp version of the OSS macro package, `Ehash` is rather inefficient, because Common Lisp does not provide incremental support for scanning the elements of a hash table. However, in the Symbolics Common Lisp version of the OSS macro package, `Ehash` is quite efficient.

- `Esymbols &optional (`*package* `*package*`*`)`* ⇒ *symbols*

  This function creates an OSS series of the symbols in *package* (which defaults to the current package). (There are no guarantees as to the order in which symbols will be enumerated.)

```
(Esymbols) ⇒ [foo bar baz ... zot] ;in some order
```

  In the pure Common Lisp version of the OSS macro package, `Esymbols` is rather inefficient, because Common Lisp does not provide incremental support for scanning the symbols in a package. However, in the Symbolics Common Lisp version of the OSS macro package, `Esymbols` is quite efficient.

- `Efile` *name* ⇒ *items*

  This function creates an OSS series of the items written in the file named *name*. The function combines the functionality of `with-open-file` with the action of reading from the file (using `read`). It is guaranteed that the file will be closed correctly, even if an error occurs. As an example of using `Efile`, assume that the forms `(a)`, `(1 2)`, and `T` have been written into the file `"test.lisp"`.

```
(Efile "test.lisp") ⇒ [(a) (1 2) T]
```

- `EnumerateF` *init step* `&optional` *test* ⇒ *items*

  The higher-order function `EnumerateF` is used to create new kinds of enumerators. The *init* must be a value of some type *T1*. The *step* argument must be a non-OSS function from *T1* to *T1*. The *test* argument (if present) must be a non-OSS function from *T1* to boolean.

  Suppose that the series returned by `EnumerateF` is S. The first output element S_0 has the value $S_0 = init$. For subsequent elements, $S_i = step(S_{i-1})$.

  If the *test* is present, the output consists of elements up to, but not including, the first element for which $test(S_i)$ is true. In addition, it is guaranteed that *step* will not be applied to the element for which *test* is true. If there is no *test*, then the output series will be of unbounded length. (In this situation, `EnumerateF` is not an early terminator.)

```
(EnumerateF '(a b c d) #'cddr #'null) ⇒ [(a b c d) (c d)]
(EnumerateF '(a b c d) #'cddr) ⇒ [(a b c d) (c d) nil nil ...]
(EnumerateF list #'cdr #'null) ≡ (Esublists list)
```

  If there is no *test*, then each time an element is output, the function *step* is applied to it. Therefore, it is important that other factors in an expression cause termination before `EnumerateF` computes an element which *step* cannot be applied to. In this regard, it is interesting that the following equivalence is almost, but not quite true. The difference is that including the *test* argument in the call on `EnumerateF` guarantees that *step* will not be applied to the element which fails *test*, while the expression using `TuntilF` guarantees that it will.

```
(TuntilF test (EnumerateF init step)) ≢ (EnumerateF init step test)
```

- `Enumerate-inclusiveF` *init step test* ⇒ *items*

  The higher-order function `Enumerate-inclusiveF` is the same as `EnumerateF` except that the first element for which *test* is true is included in the output. As with `EnumerateF`, it is guaranteed that *step* will not be applied to the element for which *test* is true.

```
(Enumerate-inclusiveF '(a b) #'cddr #'null) ⇒ [(a b) ()]
```

On-Line Transducers

Transducers compute OSS series from OSS series and form the heart of most OSS expressions. This section and the next one present the predefined transducers that are on-line (i.e., all of their inputs and outputs are on-line). These transducers are singled out because they can be used more flexibly than the transducers which are off-line. In particular, it is impossible to violate the off-line port isolation restriction without using an off-line transducer.

- `Tprevious` *items* `&optional` (*default* `nil`) (*amount* `1`) ⇒ *shifted-items*

  This function creates a series which is shifted right *amount* elements. The input *amount* must be a positive integer. The shifting is done by inserting *amount* copies of *default* before *items* and discarding *amount* elements from the end of *items*. The output is always the same length as the input.

```
(Tprevious [a b c]) ⇒ [nil a b]
(Tprevious [a b c] 'z) ⇒ [z a b]
(Tprevious [a b c] 'z 2) ⇒ [z z a]
(Tprevious []) ⇒ []
```

  The word *previous* is used as the root for the name of this function, because the function is typically used to access previous values of a series. An example of `Tprevious` used in this way is shown in conjunction with `Tuntil` below.

  To insert some amount of stuff in front of a series without losing any of the elements off the end, use `Tconcatenate` as shown below.

```
(Tconcatenate [z z] [a b c]) ⇒ [z z a b c]
```

- `Tlatch` *items* `&key :after :before :pre :post` ⇒ *masked-items*

  This function acts like a *latch* electronic circuit component. Each input element causes the creation of a corresponding output element. After a specified number of non-null input elements have been encountered, the latch is triggered and the output mode is permanently changed.

  The `:after` and `:before` arguments specify the latch point. The latch point is just after the `:after`-th non-null element in *items* or just before the `:before`-th non-null element. If neither `:after` nor `:before` is specified, an `:after` of 1 is assumed. If both are specified, it is an error.

  If a `:pre` is specified, every element prior to the latch point is replaced by this value. If a `:post` is specified, this value is used to replace every element after the latch point. If neither is specified, a `:post` of `nil` is assumed.

```
(Tlatch [nil c nil d e]) ⇒ [nil c nil nil nil]
(Tlatch [nil c nil d e] :before 2 :post T) ⇒ [nil c nil T T]
(Tlatch [nil c nil d e] :before 2 :pre 'z) ⇒ [z z z d e]
```

As a more realistic example of using `Tlatch`, suppose that a programmer wants to write a program `get-codes` which takes in a list and returns a list of all of the numbers which appear in the list after the second number in the list.

```
(defun get-codes (list)
  (letS ((elements (Elist list)))
    (Rlist (Tselect (Tlatch (numberp elements) :after 2 :pre nil)
                    elements))))
(get-codes '(a b 3 4 c d 5 e 6 f)) ⇒ (5 6)
```

- `Tuntil` *bools items* ⇒ *initial-items*

This function truncates an OSS series of elements based on an OSS series of boolean values. The output consists of all of the elements of *items* up to, but not including, the first element which corresponds to a non-null element of *bools*. That is to say, if the first non-null value in *bools* is the mth, the output will consist of all of the elements of *items* up to, but not including, the mth. (The effect of including the mth element in the output can be obtained by using `Tprevious` as shown in the last example below.) In addition, the output terminates as soon as either input runs out of elements even if a non-null element of *bools* has not been encountered.

```
(Tuntil [nil nil T nil T] [1 2 -3 4 -5]) ⇒ [1 2]
(Tuntil [nil nil T nil T] [1]) ⇒ [1]
(Tuntil (Eoss :R nil) (Eup)) ⇒ [0 1 2 ...]
(Tuntil [nil nil T nil T] (Eup)) ⇒ [0 1]
(letS ((x [1 2 -3 4 -5]))
  (Tuntil (minusp x) x)) ⇒ [1 2]
(letS ((x [1 2 -3 4 -5]))
  (Tuntil (Tprevious (minusp x)) x)) ⇒ [1 2 -3]
```

If the *items* input of `Tuntil` is such that it can be used as a destination for `alterS`, then the output of `Tuntil` can be used as a destination for `alterS`.

```
(letS* ((list '(a b 10 c))
        (x (Elist list))
        (y (Tuntil (numberp x) x)))
  (alterS y (list y))
  list) ⇒ ((a) (b) 10 c)
```

- `TuntilF` *pred items* ⇒ *initial-items*

This function is the same as `Tuntil` except that it takes a functional argument instead of an OSS series of boolean values. The non-OSS function *pred* is mapped over *items* in order to obtain a series of boolean values. (Like `Tuntil`, `TuntilF` is can be used as a destination of `alterS` if *items* can.) The basic relationship between `TuntilF` and `Tuntil` is shown in the last example below.

```
(TuntilF #'minusp [1 2 -3 4 -5]) ⇒ [1 2]
(TuntilF #'minusp [1]) ⇒ [1]
(TuntilF #'minusp (Eup)) ⇒ [0 1 2 ...]
(TuntilF pred items)
  ≡ (letS ((var items)) (Tuntil (TmapF pred var) var))
```

The functions `Tuntil` and `TuntilF` are both early terminators. This can sometimes lead to conflicts with the restriction that within each on-line subexpression, there must be a data flow path from each termination point to each output. To get the same effect without using an early terminator use `Tselect` of `Tlatch` as shown below.

```
(Tuntil bools items)
  ≡ (Tselect (not (Tlatch bools :post T)) items)

(TuntilF #'pred items)
  ≡ (Tselect (not (Tlatch (pred items) :post T)) items)
```

- `TmapF` *function* `&rest` *items-list* ⇒ *items*

The higher-order function `TmapF` is used to create simple kinds of on-line transducers. Its arguments are a single function and zero or more OSS series. The *function* argument must be a non-OSS function which is compatible with the number of input series and the types of their elements.

A single OSS series is returned. Each element of this series is the result of applying *function* to the corresponding elements of the input series. (That is to say, if `TmapF` receives a single input series R it will return a single output S such that $S_i = function(R_i)$.) The length of the output is the same as the length of the shortest input. If there are no bounded series inputs (e.g., if there are no series inputs), then `TmapF` will generate an unbounded OSS series.

```
(TmapF #'+ [1 2 3] [4 5]) ⇒ [5 7]
(TmapF #'sqrt []) ⇒ []
(TmapF #'gensym) ⇒ [#:G003 #:G004 #:G005 ...]
```

- `TscanF` {*init*} *function items* ⇒ *results*

The higher-order function `TscanF` is used to create complex kinds of on-line transducers. (The name is borrowed from APL.) The *init* argument (if present) must be a non-OSS value of some type *T1*. The *function* argument must be a binary non-OSS function from *T1* and some type *T2* to *T1*. The *items* argument must be an OSS series whose elements are of type *T2*. If the *init* argument is not present than *T1* must equal *T2*.

The *function* argument is used to compute a series of accumulator values of type *T1* which is returned as the output of `TscanF`. The output is the same length as the series input and consists of the successive accumulator values.

Suppose that the series input to `TscanF` is R and the output is S. The basic relationship between the output and the input is that $S_i = function(S_{i-1}, R_i)$. If the *init* argument is specified, it is used as an initial value of the accumulator and the first output element S_0 has the value $S_0 = function(init, R_0)$. Typically, but not necessarily, *init* is chosen so that it is a left identity of *function*. If that is the case, then $S_0 = R_0$. It is important to remember that the elements of *items* are used as the second argument of *function*. The order of arguments is chosen to highlight this fact.

```
(TscanF 0 #'+ [1 2 3]) ⇒ [1 3 6]
(TscanF 10 #'+ [1 2 3]) ⇒ [11 13 16]
(TscanF nil #'cons [a b]) ⇒ [(nil . a) ((nil . a) . b)]
(TscanF nil #'(lambda (state x) (cons x state)) [a b]) ⇒ [(a) (b a)]
```

If the *init* argument is not specified, then the first element of the output is computed differently from the succeeding elements and $S_0 = R_0$. (If *function* is cheap to evaluate, `TscanF` runs more efficiently if it is provided with an *init* argument.) One situation where one typically has to leave out the *init* argument is when *function* does not have a left identity element as in the last example below.

```
(TscanF #'+ [1 2 3]) ⇒ [1 3 6]
(TscanF #'max [1 3 2]) ⇒ [1 3 3]
```

An interesting example of a scanning process is the operation of *proration*. In this process, a total is divided up and allocated between a number of categories. The allocation is done based on percentages which are associated with the categories. (For example, some number of packages might be divided up between a number of people.) One might think that this could be done straightforwardly by multiplying the total by each of the percentages. Unfortunately, this mapping approach does not work.

The proration problem is more complex than it first appears. Typically, there is a limit to the divisibility of the total (e.g., when a group of packages is divided up, the individual packages cannot be subdivided). This means that rounding must be performed each time the total is multiplied by a percentage. In addition, it is usually important that the total be allocated exactly—i.e., that the sum of the allocations be exactly equal to the total, rather than being one more or one less. Scanning is required in order to make sure that things come out exactly right.

As a concrete example of proration, suppose that 99 packages need to be allocated among three people based on the percentages 35%, 45%, and 20%. Assuming that the percentages and the number of packages are all represented as integers, simple mapping would lead to the incorrect result below in which the allocations add up to 100 instead of 99.

```
(prognS (round (/ (* 99 [35 45 20]) 100))) ⇒ [35 45 20]
```

The transducer `Tprorate` below solves the proration problem by using `TscanF`. It takes in a total and an OSS series of percentages and returns an OSS series of allocations. The basic action of the program is to multiply each percentage by the total. However, it also keeps track of how much of the total has been allocated. When the last percentage is encountered, the allocation is set to be everything which remains to be allocated. (This can cause a significant distortion in the final allocation, but it guarantees that the allocations will always add up to the total no matter what has happened with rounding along the way.) In order to determine when the last percentage is being encountered, the program keeps track of how much percentage has been accounted for and assumes that the percentages always add up to 100.

```
(defun prorate-step (state percent)
  (let* ((total (second state))
         (unallocated (third state))
         (unused-percent (fourth state))
         (allocation (if (= percent unused-percent) unallocated
                         (round (/ (* total percent) 100)))))
    (setf (first state) allocation)
    (setf (third state) (- unallocated allocation))
    (setf (fourth state) (- unused-percent percent))
    state))

(defunS Tprorate (total percents)
    (declare (type oss percents))
  (car (TscanF (list 0 total total 100) #'prorate-step percents)))

(Tprorate 99 [35 45 20]) ⇒ [35 45 19]
```

An interesting aspect of the function `Tprorate` is that the state manipulated by the scanned function `prorate-step` has four parts: an allocation, the total, the unallocated portion of the total, and the remaining percentage not yet allocated. This illustrates the fact that `TscanF` can be used with complex state objects. (The same is true of `EnumerateF` and `ReduceF`.) However, it also illustrates that accessing the various parts of a complex state is awkward and inefficient.

Fortunately, it is often possible to get around the need for a complex state object by using a compound OSS expression. For the example of proration, this can be done as shown below. Simple mapping is combined with two scans which keep track of cumulative values. An implicitly mapped test is used to make sure that things come out right on the last step. (The function `Tprevious` is used to access the previous value of the series `unallocated`.)

```
(defunS Tprorate-multi-state (total percents)
    (declare (type oss percents))
  (letS* ((allocation (round (/ (* percents total) 100)))
          (unallocated (TscanF total #'- allocation))
          (unused-percent (TscanF 100 #'- percents)))
    (if (zerop unused-percent)
        (Tprevious unallocated total)
        allocation)))
```

Cotruncation

A key feature of every on-line transducer is that it terminates as soon as any input runs out of elements. Put another way, the output is never longer than the shortest input. (If the transducer is also an early terminator, then the output can be shorter than the shortest input, otherwise it must be the same length as the shortest input.) This effect is referred to as cotruncation, because it acts as if each input had been truncated to the length of the shortest input. If several enumerators and on-line transducers are combined together into an OSS expression, cotruncation will typically cause all of the series produced by the enumerators to be truncated to the same length. For example, in the expression below, all of the series (including the unbounded series produced by `Eup`) are truncated to a length of two.

```
(Rlist (* (+ (Eup) [4 5]) [1 2 3])) ⇒ (4 12)
```

- `Tcotruncate` *items* `&rest` *more-items* ⇒ *initial-items* `&rest` *more-initial-items*

It is occasionally important to specify cotruncation explicitly. This can be done with the function `Tcotruncate` whose only action is to force all of the outputs to be of the same length. (If any of the inputs of `Tcotruncate` are such that they can be used as destinations of `alterS`, then the corresponding outputs of `Tcotruncate` can be used as destinations of `alterS`.)

```
(Tcotruncate [1 2 -3 4 -5] [10]) ⇒ [1] [10]
(Tcotruncate (Eup) [a b]) ⇒ [0 1] [a b]
(Tcotruncate [a b] []) ⇒ [] []
```

An important feature of `Tcotruncate` is that it has a powerful interaction with the requirement that within each on-line subexpression, there must be a data flow path from each termination point to each output. Consider the function `weighted-squares` below. This program is intended to take a vector of values and a vector of weights and return a list of two vectors: the squares of the values and the squares multiplied by the weights. The program violates the requirement above, because there is no data flow path from `(Evector weight-vector)` to `(Rvector squares)`.

```
(defun weighted-squares (value-vector weight-vector)
  (letS* ((values (Evector value-vector))       ;Signals warning 18
          (weights (Evector weight-vector))
          (squares (* values values))
          (weighted-squares (* squares weights)))
    (list (Rvector squares) (Rvector weighted-squares))))

(weighted-squares #(1 2 3) #(2 3 4)) ⇒ (#(1 4 9) #(2 12 36))
(weighted-squares #(1 2) #(2 3 4)) ⇒ (#(1 4) #(2 12))
(weighted-squares #(1 2 3) #(2 3)) ⇒ (#(1 4 9) #(2 12))
```

It might be the case that the programmer knows that `value-vector` and `weight-vector` always have the same length. (Or it might be the case that he wants both output values to be no longer than the shortest input.) In either case, the function can be written as shown below which is much more efficient than the program above since there is no longer a restriction violation which triggers code copying. The key difference is that the use of `Tcotruncate` makes both outputs depend on both inputs. If the inputs are known to be the same length, the use of `Tcotruncate` can be thought of as a declaration.

```
(defun weighted-squares* (value-vector weight-vector)
  (letS* (((values weights)
           (Tcotruncate (Evector value-vector)
                        (Evector weight-vector)))
          (squares (* values values))
          (weighted-squares (* squares weights)))
    (list (Rvector squares) (Rvector weighted-squares))))

(weighted-squares* #(1 2 3) #(2 3 4)) ⇒ (#(1 4 9) #(2 12 36))
(weighted-squares* #(1 2) #(2 3 4)) ⇒ (#(1 4) #(2 12))
(weighted-squares* #(1 2 3) #(2 3)) ⇒ (#(1 4) #(2 12))
```

Off-Line Transducers

This section and the next two describe transducers that are not on-line. Most of these functions have some inputs or outputs which are on-line. The ports which are on-line can be used freely. However, the off-line ports have to be isolated when they are used. (For ease of reference, the off-line ports all begin with the letter code "*O*".)

- `Tremove-duplicates` *Oitems* `&optional` (*comparator* `#'eql`) ⇒ *items*

  This function is analogous to `remove-duplicates`. It creates an OSS series that has the same elements as the off-line input *Oitems* with all duplicates removed. The *comparator* is used to determine whether or not two items are duplicates. If two items are the same, then the item which is later in the series is discarded. (As in `remove-duplicates` the algorithm employed is not particularly efficient, being $O(n^2)$.) (If the *Oitems* input of `Tremove-duplicates` is such that it can be used as a destination for `alterS`, then the output of `Tremove-duplicates` can be used as a destination for `alterS`.)

```
(Tremove-duplicates [1 2 1 (a) (a)]) ⇒ [1 2 (a) (a)]
(Tremove-duplicates [1 2 1 (a) (a)] #'equal) ⇒ [1 2 (a)]
```

- `Tchunk` *amount Oitems* ⇒ *lists*

  This function creates an OSS series of lists of length *amount* of successive subseries of the off-line input *Oitems*. If the length of *Oitems* is not a multiple of *amount*, then the last (`mod (Rlength` *Oitems*) *amount*) elements of *Oitems* will not appear in any output chunk.

```
(Tchunk 2 [a b c d e]) ⇒ [(a b) (c d)]
(Tchunk 6 [a b c d]) ⇒ []
```

- `Twindow` *amount Oitems* ⇒ *lists*

  This function creates an OSS series of lists of length *amount* of subseries of the off-line input *Oitems* starting at each element position. If the length of *Oitems* is less than *amount*, the output will not contain any windows. The last example below shows `Twindow` being used to compute a moving average.

```
(Twindow 2 [a b c d]) ⇒ [(a b) (b c) (c d)]
(Twindow 4 [a b c d]) ⇒ [(a b c d)]
(Twindow 6 [a b c d]) ⇒ []
(prognS (/ (apply #'+ (Twindow 2 [2 4 6 8])) 2)) ⇒ [3 5 7]
```

- `Tconcatenate` *Oitems1 Oitems2* `&rest` *more-Oitems* ⇒ *items*

  This function creates an OSS series by concatenating together two or more off-line input OSS series. The length of the output is the sum of the lengths of the inputs. (The elements of the individual input series are not computed until they need to be.)

```
(Tconcatenate [b c] [] [d]) ⇒ [b c d]
(Tconcatenate [] []) ⇒ []
```

- `TconcatenateF` *Enumerator Oitems* ⇒ *items*

  The *Enumerator* must be a quoted OSS function that is an enumerator. The function `TconcatenateF` applies *Enumerator* to each element of the off-line input *Oitems* and returns the series obtained by concatenating all of the results together. If *Enumerator* returns multiple values, then `TconcatenateF` will as well.

```
(TconcatenateF #'Elist [(a b) () (c d)]) ⇒ [a b c d]
(TconcatenateF #'Elist [() ()]) ⇒ []
(TconcatenateF #'Eplist [(a 1) (b 2 c 3)]) ⇒ [a b c] [1 2 3]
```

- `Tsubseries` *Oitems start* `&optional` *below* ⇒ *items*

  This function creates an OSS series containing a subseries of the elements of the off-line input *Oitems* from *start* up to, but not including, *below*. If `below` is greater than the length of *Oitems*, output nevertheless stops as soon as the input runs out of elements. If *below* is not specified, the output continues all the way to the end of *Oitems*. Both of the arguments *start* and *below* must be non-negative integers.

```
(Tsubseries [a b c d] 1) ⇒ [b c d]
(Tsubseries [a b c d] 1 3) ⇒ [b c]
(Rlist (Tsubseries (Elist list) 1 2)) ≡ (subseq list 1 2)
```

  If the *Oitems* input of `Tsubseries` is such that it can be used as a destination for `alterS`, then the output of `Tsubseries` can be used as a destination for `alterS`.

```
(let ((list '(a b c d e)))
  (alterS (Tsubseries (Elist list) 1 3) (Eup))
  list) ⇒ (a 0 1 d e)
```

  The function `Tsubseries` terminates as soon as it has written the last output element. As a result, it is an early terminator. This can sometimes lead to conflicts with the restriction that within each on-line subexpression, there must be a data flow path from each termination point to each output. To select a subseries without using an early terminator, use `Tselect`, `Tmask`, and `Eup` as shown below.

```
(Tsubseries Oitems from below)
  ≡ (Tselect (Tmask (Eup from :below below)) Oitems)
```

- `Tpositions` *Obools* ⇒ *indices*

  This function takes in an OSS series and returns an OSS series of the indexes of the non-null elements in the off-line input series.

```
(Tpositions [T nil T 44]) ⇒ [0 2 3]
(Tpositions [nil nil nil]) ⇒ []
```

- `Tmask` *Omonotonic-indices* ⇒ *bools*

  This function is a quasi-inverse of `Tpositions`. The input *Omonotonic-indices* must be a strictly increasing OSS series of non-negative integers. The output, which is always unbounded, contains `T` in the positions specified by *Omonotonic-indices* and `nil` everywhere else.

```
(Tmask [0 2 3]) ⇒ [T nil T T nil nil ...]
(Tmask []) ⇒ [nil nil ...]
(Tmask (Tpositions x)) ≡ (Tconcatenate (not (null x)) (Eoss :R nil))
```

- `Tmerge` *Oitems1 Oitems2 comparator* ⇒ *items*

  This function is analogous to `merge`. The output series contains the elements of the two off-line input series. The elements of *Oitems1* appear in the same order that they are read in. Similarly, the elements of *Oitems2* appear in the same order that they are read in. However the elements from the two inputs are intermixed under the control of the *comparator*. At each step, the *comparator* is used to compare the current elements in the two series. If the *comparator* returns non-null, the current element is removed from *Oitems1* and transferred to the output. Otherwise, the next output comes from *Oitems2*. (If, as in the first example below, the elements of the individual input series are ordered with respect to *comparator*, then the result will also be ordered with respect to *comparator*. If, as in the second example below, either input is not ordered, the result will not be ordered.)

```
(Tmerge [1 3 7 9] [4 5 8] #'<) ⇒ [1 3 4 5 7 8 9]
(Tmerge [1 7 3 9] [4 5 8] #'<) ⇒ [1 4 5 7 3 8 9]
(Tmerge x y #'(lambda (x y) T)) ≡ (Tconcatenate x y)
```

- `Tlastp` *Oitems* ⇒ *bools items*

  This function takes in a series and returns a series of boolean values having the same length such that the last value is `T` and all of the other values are `nil`. If the input series is unbounded, then the output series will also be unbounded and every element of the output will be `nil`.

  It turns out that this output cannot be computed by an on-line OSS function. Therefore, if `Tlastp` returned only the boolean values described above, the isolation restrictions would make it impossible to use the input series and the output values together in the same computation. In order to get around this problem, `Tlastp` returns a second output which is identical to the input. This output can be used in lieu of the input in combination with the boolean values.

```
(Tlastp [a b c d]) ⇒ [nil nil nil T] [a b c d]
(Tlastp [a]) ⇒ [T] [a]
(Tlastp []) ⇒ [] []
(Tlastp (Eup)) ⇒ [nil nil nil ...] [0 1 2 ...]
```

As an example of using `Tlastp`, it is interesting to return to the example of proration discussed in conjunction with the function `TscanF`. Both of the proration functions presented earlier assume that the percentages always add up to 100. If this turns out not to be the case, then an exact allocation of the total is not guaranteed. The following program ensures that exact allocation will occur no matter what the percentages add up to. It does this by using `Tlastp` to detect which percentage is the last one.

```
(defunS Tprorate-robust (total Opercents)
    (declare (type oss Opercents))
  (letS* (((is-last percents) (Tlastp Opercents))
          (allocation (round (/ (* percents total) 100)))
          (unallocated (TscanF total #'- allocation)))
    (if is-last (Tprevious unallocated total) allocation)))

(Tprorate-robust 99 [35 45 20]) ⇒ [35 45 19]
(Tprorate-robust 99 [35 45 21]) ⇒ [35 45 19]
(Tprorate 99 [35 45 21]) ⇒ [35 45 21]
```

Selection and Expansion

Selection and its inverse are particularly important kinds of off-line transducers.

- `Tselect` *bools* `&optional` *items* ⇒ *Oitems*

  This function selects elements from a series based on a boolean series. The off-line output consists of the elements of *items* which correspond to non-null elements of *bools*. That is to say, the nth element of *items* is in the output iff the nth element of *bools* is non-null. The order of the elements in *Oitems* is the same as the order of the elements in *items*. The output terminates as soon as either input runs out of elements. If no *items* input is specified, then the non-null elements of *bools* are themselves returned as the output of `Tselect`. (If the *items* input of `Tselect` is such that it can be used as a destination for `alterS`, then the output of `Tselect` can be used as a destination for `alterS`.)

```
(Tselect [T nil T nil] [a b c d]) ⇒ [a c]
(Tselect [a nil b nil]) ⇒ [a b]
(Tselect [nil nil] [a b]) ⇒ []
```

An interesting aspect of `Tselect` is that the output series is off-line rather than having the two input series be off-line. This is done in recognition of the fact that the two input series are always in synchrony with each other. Having only one port which is off-line allows more flexibility then having two ports which are off-line.

One might want to select elements out of a series based on their positions in the series rather than on boolean values. This can be done straightforwardly using `Tmask` as shown below.

```
(Tselect (Tmask [0 2]) [a b c d]) ⇒ [a c]
(Tselect (not (Tmask [0 2])) (Eup 10)) ⇒ [11 13 14 15 ...]
```

A final feature of `Tselect` in particular, and off-line ports in general, is illustrated by the program below. In this program, the `Tselect` causes the first `Elist` to get out of phase with the second `Elist`. As a result, it is important to think of OSS expressions as passing around series objects rather than as merely being abbreviations for loops where things are always happening in lock step. The latter point of view might lead to the idea that the output of the program below would be `((a 1) (c 2) (d 4))`.

```
(letS ((tag (Elist '(a b c d e)))
       (x (Elist '(1 -2 2 4 -5))))
  (Rlist (list tag (Tselect (plusp x) x)))) ⇒ ((a 1) (b 2) (c 4))
```

- `TselectF` *pred Oitems* ⇒ *items*

  This function is the same as `Tselect`, except that it maps the non-OSS function *pred* over *Oitems* to obtain a series of boolean values with which to control the selection. In addition, `TselectF` has an off-line input rather than an off-line output (this is fractionally more efficient). The logical relationship between `Tselect` and `TselectF` is shown in the last example below.

```
(TselectF #'identity [a nil nil b nil]) ⇒ [a b]
(TselectF #'plusp [-1 2 -3 4]) ⇒ [2 4]
(TselectF pred items)
  ≡ (letS ((var items)) (Tselect (TmapF pred var) var))
```

- `Texpand` *bools Oitems* `&optional` (*default* `nil`) ⇒ *items*

  This function is a quasi-inverse of `Tselect`. (The name is borrowed from APL.) The output contains the elements of *Oitems* spread out into the positions specified by the non-null elements in *bools*—i.e., the nth element of *Oitems* is in the position occupied by the nth non-null element in *bools*. The other positions in the output are occupied by *default*. The output stops as soon as *bools* runs out of elements, or a non-null element in *bools* is encountered for which there is no corresponding element in *Oitems*.

```
(Texpand [nil T nil T T] [a b c]) ⇒ [nil a nil b c]
(Texpand [nil T nil T T] [a]) ⇒ [nil a nil]
(Texpand [nil T] [a b c] 'z) ⇒ [z a]
(Texpand [nil T nil T T] []) ⇒ [nil]
```

Splitting

An operation which is closely related to selection, is splitting. In selection, specified elements are selected out of a series. It is not possible to apply further operations to the elements which are not selected, because they have been discarded. In contrast, splitting divides up a series into two or more parts which can be individually used. Both `Tsplit` and `TsplitF` have on-line inputs and off-line outputs. The outputs have to be off-line, because they are inherently non-synchronized with each other.

- `Tsplit` *items bools* `&rest` *more-bools* ⇒ *Oitems1 Oitems2* `&rest` *more-Oitems*

  This function takes in a series of elements and partitions them between two or more outputs. If there are n boolean inputs then there are $n+1$ outputs. Each input element is placed in exactly one output series. Suppose that the nth element of *bools* is non-null. In this case, the nth element of *items* will be placed in *Oitems1*. On the other hand, if the nth element of *bools* is `nil`, the second boolean input (if any) is consulted in order to see whether the input element should be placed in the second output or in a later output. (As in a `cond`, each time a boolean element is `nil`, the next boolean series is consulted.) If the nth element of every boolean series is `nil`, then the nth element of *items* is placed in the last output.

```
(Tsplit [-1 -2 3 4] [T T nil nil]) ⇒ [-1 -2] [3 4]
(Tsplit [-1 -2 3 4] [T T nil nil] [nil T nil T]) ⇒ [-1 -2] [4] [3]
(Tsplit [-1 -2 3 4] [T T T T]) ⇒ [-1 -2 3 4] []
```

If the *items* input of `Tsplit` is such that it can be used as a destination for `alterS`, then all of the outputs of `Tsplit` can be used as destinations for `alterS`.

```
(letS* ((list '(-1 2 -3))
        (x (Elist list))
        ((x+ x-) (Tsplit x (plusp x))))
  (alterS x+ (+ x+ 10))
  (alterS x- (- x- 10))
  list) ⇒ (-11 12 -13)
```

- `TsplitF` *items pred* `&rest` *more-pred* ⇒ *Oitems1 Oitems2* `&rest` *more-Oitems*

  This function is the same as `Tsplit`, except that it takes predicates as arguments rather than boolean series. The predicates must be non-OSS functions and are applied to *items* in order to create boolean values. The relationship between `TsplitF` and `Tsplit` is almost but not exactly as shown below.

```
(TsplitF items pred1 pred2)
  ≢ (letS ((var items))
      (Tsplit var (TmapF pred1 var) (TmapF pred2 var)))
```

The reason that the equivalence above does not quite hold is that, as in a `cond`, the predicates are not applied to individual elements of *items* unless the resulting value is needed in order to determine which output series the element should be placed in (e.g., if the first predicate returns non-null when given the nth element of items, the second predicate will not be called). This promotes efficiency and allows earlier predicates to act as guards for later predicates.

```
(TsplitF [-1 -2 3 4] #'minusp) ⇒ [-1 -2] [3 4]
(TsplitF [-1 -2 3 4] #'minusp #'evenp) ⇒ [-1 -2] [4] [3]
```

Reducers

Reducers produce non-OSS outputs based on OSS inputs. There are two basic kinds of reducers: ones that combine the elements of OSS series together into aggregate data structures (e.g., into a list) and ones that compute some summary value from these elements (e.g., the sum). All the predefined reducers are on-line. A few reducers are also early terminators. These reducers are described in the next section.

- `Rlist` *items* ⇒ *list*

  This function creates a list of the elements in *items* in order.

```
(Rlist [a b c]) ⇒ (a b c)
(Rlist []) ⇒ ()
(Rlist (fn (Elist x) (Elist y))) ≡ (mapcar #'fn x y)
(Rlist (fn (Esublists x) (Esublists y))) ≡ (maplist #'fn x y)
```

- `Rbag` *items* ⇒ *list*

  This function creates a list of the elements in *items* with no guarantees as to the order of the elements. The function `Rbag` is more efficient than `Rlist`.

```
(Rbag [a b c]) ⇒ (c a b) ;in some order
(Rbag []) ⇒ ()
```

- `Rappend` *lists* ⇒ *list*

  This function creates a list by appending the elements of *lists* together in order.

```
(Rappend [(a b) nil (c d)]) ⇒ (a b c d)
(Rappend []) ⇒ ()
```

- `Rnconc` *lists* ⇒ *list*

  This function creates a list by nconcing the elements of *lists* together in order. The function `Rnconc` is faster than `Rappend`, but modifies the lists in the OSS series *lists*.

```
(Rnconc [(a b) nil (c d)]) ⇒ (a b c d)
(Rnconc []) ⇒ ()
(let ((x '(a b))) (Rnconc (Eoss x x))) ⇒ (a b a b a b ...)
(Rnconc (fn (Elist x) (Elist y))) ≡ (mapcan #'fn x y)
(Rnconc (fn (Esublists x) (Esublists y))) ≡ (mapcon #'fn x y)
```

- `Ralist` *keys values* ⇒ *alist*

  This function creates an alist containing *keys* and *values*. It terminates as soon as either of the inputs runs out of elements. If there are duplicate keys, they will be put on the alist, but order is preserved.

```
(Ralist [a b] [1 2]) ⇒ ((a . 1) (b . 2))
(Ralist [a b] []) ⇒ ()
(Ralist keys values) ≡ (Rlist (cons keys values))
```

- `Rplist` *indicators values* ⇒ *plist*

  This function creates a plist containing *keys* and *values*. It terminates as soon as either of the inputs runs out of elements. If there are duplicate indicators, they will be put on the plist, but order is preserved.

```
(Rplist [a b a] [1 2 3]) ⇒ (a 1 b 2 a 3)
(Rplist [a b] []) ⇒ ()
(Rplist keys values) ≡ (Rnconc (list keys values))
```

- `Rhash` *keys values* `&rest` *option-plist* ⇒ *table*

  This function creates a hash table containing *keys* and *values*. It terminates as soon as either of the inputs runs out of elements. The *option-plist* can contain any options acceptable to `make-hash-table`. The *option-plist* cannot refer to variables bound by `letS`.

```
(Rhash [color name] [brown fred]) ⇒ #<hash-table 23764432>
;;hash table containing color->brown, name->fred
(Rhash [color name] []) ⇒ #<hash-table 23764464>
;;empty hash table
```

- `Rvector` *items* `&key :size &rest` *option-plist* ⇒ *vector*

This function creates a vector containing the elements of *items* in order. The *option-plist* can contain any options acceptable to `make-array`. The *option-plist* cannot refer to variables bound by `letS`.

The function `Rvector` operates in one of two ways. If the `:size` argument is supplied, then `Rvector` assumes that *items* will contain exactly `:size` elements. A vector is created of length `:size` with the options specified in *option-plist* and the elements of *items* are stored in it. (If *items* has fewer than `:size` elements, some of the slots in the vector will be left in their initial state. If *items* has more than `:size` elements, an error will ensue.) In this mode, `Rvector` is very efficient, but rather inflexible.

```
(Rvector [1 2 3] :size 3) ⇒ #(1 2 3)
(Rvector [#\B #\A #\R] :size 3 :element-type 'string-char) ⇒ "BAR"
(Rvector [1] :size 4 :initial-element 0) ⇒ #(1 0 0 0)
```

If the `:size` argument is not supplied, then `Rvector` allows for the creation of an arbitrarily large vector. It does this by using `vector-push-extend`. In order for this to work, it forces `:adjustable` to be `T` and `:fill-pointer` to be 0 no matter what is specified in the *options-list*. In this mode, an arbitrary number of input elements can be handled, however, things are much less efficient, since the vector created is not a simple vector.

```
(Rvector [1 2 3]) ⇒ #(1 2 3)
(Rvector []) ⇒ #()
(Rvector [#\B #\A #\R] :element-type 'string-char) ⇒ "BAR"
```

To store a series in a preexisting vector, use `alterS` of `Evector`.

```
(let ((v '#(a b c)))
  (alterS (Evector v) (Eoss 1 2))
  v) ⇒ #(1 2 c)
```

- `Rfile` *name items* `&rest` *option-plist* ⇒ `T`

This function creates a file named *name* and writes the elements of *items* into it using `print`. The *option-plist* can contain any of the options accepted by `open` except `:direction` which is forced to be `:output`. All of the ordinary printer control variables are obeyed during the printout. The value `T` is always returned. The *option-plist* cannot refer to variables bound by `letS`.

```
(Rfile "test.lisp" ['(a) '(1 2) T] :if-exists :append) ⇒ T
;;The output "
;;(a)
;;(1 2)
;;T " is printed into the file "test.lisp".
```

- `Rlast` *items* `&optional (`*default* `nil)` ⇒ *item*

This function returns the last element of *items*. If *items* is of zero length, *default* is returned.

```
(Rlast [a b c]) ⇒ c
(Rlast [] 'z) ⇒ z
```

- `Rlength` *items* ⇒ *number*

  This function returns the number of elements in *items*.

```
(Rlength [a b c]) ⇒ 3
(Rlength []) ⇒ 0
```

- `Rsum` *numbers* ⇒ *number*

  This function computes the sum of the elements in *numbers*. These elements must be numbers, but they need not be integers.

```
(Rsum [1 2 3]) ⇒ 6
(Rsum []) ⇒ 0
(Rsum [1.1 1.2 1.3]) ⇒ 3.6
```

- `Rmax` *numbers* ⇒ *number*

  This function computes the maximum of the elements in *numbers*. These elements must be non-complex numbers, but they need not be integers. The value `nil` is returned if *numbers* has length zero.

```
(Rmax [2 1 4 3]) ⇒ 4
(Rmax []) ⇒ nil
(Rmax [1.2 1.1 1.4 1.3]) ⇒ 1.4
```

- `Rmin` *numbers* ⇒ *number*

  This function computes the minimum of the elements in *numbers*. These elements must be non-complex numbers, but they need not be integers. The value `nil` is returned if *numbers* has length zero.

```
(Rmin [2 1 4 3]) ⇒ 1
(Rmin []) ⇒ nil
(Rmin [1.2 1.1 1.4 1.3]) ⇒ 1.1
```

- `ReduceF` *init function items* ⇒ *result*

  This function is analogous to `reduce`. In addition, it is similar to `TscanF` except that *init* is not optional and the final value of the accumulator is the only value returned as shown in the last example below. If *items* is of length zero, *init* is returned. As with `TscanF`, *function* must be a non-OSS function and the value of *init* is typically chosen to be a left identity of *function*. It is important to remember that the elements of *items* are used as the second argument of *function*. The order of arguments is chosen to highlight this fact.

```
(ReduceF 0 #'+ [1 2 3]) ⇒ 6
(ReduceF 0 #'+ []) ⇒ 0
(ReduceF 0 #'+ x) ≡ (Rsum x)
(ReduceF init function items)
  ≡ (letS ((var init))
      (Rlast (TscanF var function items) var))
```

  In order to do reduction without an initial seed value, use `Rlast` of `TscanF`. Note that although a seed value does not have to be specified, a value to be returned if there are no elements in *items* still has to be specified.

```
(Rlast (TscanF #'max x) nil) ≡ (Rmax x)
```

Early Reducers

The following four reducers are early terminators. Each of these functions has a non-early variant denoted by the suffix "`-late`". The early variants are more efficient, because they terminate as soon as they have determined a result. This may be long before any of the input series run out of elements. However, as discussed at the end of this section, one has to be somewhat careful when using an early reducer in an OSS expression.

- `Rfirst` *items* `&optional` (*default* `nil`) ⇒ *item*
- `Rfirst-late` *items* `&optional` (*default* `nil`) ⇒ *item*

Both of these functions return the first element of *items*. If *items* is of zero length, *default* is returned. The only difference between the functions is that `Rfirst` stops immediately after reading the first element of *items*, while `Rfirst-late` does not terminate until *items* runs out of elements.

```
(Rfirst [a b c]) ⇒ a
(Rfirst [] 'z) ⇒ z
```

- `Rnth` *n items* `&optional` (*default* `nil`) ⇒ *item*
- `Rnth-late` *n items* `&optional` (*default* `nil`) ⇒ *item*

Both of these functions return the *n*th element of *items*. If *n* is greater than or equal to the length of *items*, *default* is returned. The only difference between the functions is that `Rnth` stops immediately after reading the *n*th element of *items*, while `Rnth-late` does not terminate until *items* runs out of elements.

```
(Rnth 1 [a b c]) ⇒ b
(Rnth 1 [] 'z) ⇒ z
```

- `Rand` *bools* ⇒ *bool*
- `Rand-late` *bools* ⇒ *bool*

Both of these functions compute the `and` of the elements in *bools*. As with the function `and`, `nil` is returned if any element of *bools* is `nil`. Otherwise the last element of *bools* is returned. The value `T` is returned if *bools* has length zero. The only difference between the functions is that `Rand` terminates as soon as a `nil` is encountered in the input, while `Rand-late` does not terminate until *bools* runs out of elements.

```
(Rand [a b c]) ⇒ c
(Rand [a nil c]) ⇒ nil
(Rand []) ⇒ T
(Rand (pred (Esequence x) (Esequence y))) ≡ (every #'pred x y)
```

- `Ror` *bools* ⇒ *bool*
- `Ror-late` *bools* ⇒ *bool*

Both of these functions compute the `or` of the elements in *bools*. As with the function `or`, `nil` is returned if every element of *bools* is `nil`. Otherwise the first non-null element of *bools* is returned. The value `nil` is returned if *bools* has length zero. The only difference between the functions is that `Ror` terminates as soon as a non-null value is encountered in the input, while `Ror-late` does not terminate until *bools* runs out of elements.

```
(Ror [a b c]) ⇒ a
(Ror [a nil c]) ⇒ a
(Ror []) ⇒ nil
(Ror (pred (Esequence x) (Esequence y))) ≡ (some #'pred x y)
```

Care must be taken when using early reducers. As discussed in the section on restrictions, OSS expressions are required to obey the restriction that within each on-line subexpression, there must be a data flow path from each termination point to each output. Early reducers interact with this restriction since early reducers are termination points. As a result, there must be a data flow path from each early reducer to each output of the containing on-line subexpression.

Since reducers compute non-OSS values, they directly compute outputs of on-line subexpressions. As a result, it is impossible for there to be a data flow path from a reducer to any output other than the output the reducer itself computes. Therefore, the use of an early reducer will trigger code copying unless that reducer computes the only output of the on-line subexpression.

For example, consider the following four expressions. The first two expressions return the same result. However, the first is more efficient. This is a prototypical example of a situation where it is better to use an early reducer. In contrast, although the last two expressions also return the same results, the *second* of the expressions is more efficient. The problem is that in the first of these expressions, there is no data flow path from the use of `Rfirst` to the second output. In order to fix this problem the OSS macro package duplicates the list enumeration. It is more efficient to use a non-early reducer as in the last example.

```
(letS ((x (Elist '(1 2 -3 4 5 -6 -7 8))))
  (Rfirst (TselectF #'minusp x))) ⇒ -3

(letS ((x (Elist '(1 2 -3 4 5 -6 -7 8))))
  (Rfirst-late (TselectF #'minusp x))) ⇒ -3

(letS ((x (Elist '(1 2 -3 4 5 -6 -7 8))))   ;Signals warning 18
  (valS (Rfirst (TselectF #'minusp x))
        (Rsum x))) ⇒ -3 4

(letS ((x (Elist '(1 2 -3 4 5 -6 -7 8))))
  (valS (Rfirst-late (TselectF #'minusp x))
        (Rsum x))) ⇒ -3 4
```

Series Variables

The principal way to create OSS variables is to use the form `letS`. (These variables are also created by the forms `lambdaS` and `defunS`.)

- `letS` *var-value-pair-list* {*decl*}* `&body` *expr-list* ⇒ *result*

The form `letS` is syntactically analogous to `let`. Just as in a `let`, the first subform is a list of variable-value pairs. The `letS` form defines the scope of these variables and gives them the indicated values. As in a `let`, one or more declarations can follow the variable-value pairs. These can be used to specify the types of the variables.

The variables created by `letS` can be OSS variables or non-OSS variables. Which are which is determined by the type of the value that is bound to the variable. As in `let`, the variables are bound in parallel. In the example below, `y` is an OSS variable while `x` and `z` are non-OSS variables.

```
(letS ((x '(1 2 3))
       (y (Elist '(1 2 3)))
       (z (Rsum (Elist '(1 2 3)))))
  (list x (Rmax y) z)) ⇒ ((1 2 3) 3 6)
```

Unlike `let`, `letS` does not support degenerate variable-value pairs which consist solely of a variable. (Since `letS` variables cannot be assigned to, see below, degenerate pairs would be of little value.)

```
(letS (x) ...)                          ;Signals error 9
```

The following example illustrates the use of a declaration in a `letS`. Declarations are handled in the same way that they are handled in a `let`.

```
(letS ((x (Elist '(1 2 3))))
    (declare (type integer x))
  (Rsum x)) ⇒ 6
```

The form `letS` goes beyond `let` to include the functionality of `multiple-value-bind`. A variable in a variable-value pair can be a list of variables instead of a single variable. When this is the case, the variables pick up the first, second, etc. results returned by the value expression. (If there is only one variable, it gets the first value. If nil is used in lieu of a variable, the corresponding value is ignored.) If there are fewer variables than values, the extra values are ignored. Unlike `multiple-value-bind`, `letS` signals an error if there are more variables than values. (Note that there is no form `multiple-value-bindS` and that the form `multiple-value-bind` cannot be used inside of an OSS expression to bind the results of an OSS function.)

```
(letS (((key value) (Ealist '((a . 1) (b . 2)))))
  (Rlist (list key value))) ⇒ ((a 1) (b 2))

(letS ((key (Ealist '((a . 1) (b . 2)))))
  (Rlist key)) ⇒ (a b)

(letS (((nil value) (Ealist '((a . 1) (b . 2)))))
  (Rlist value)) ⇒ (1 2)

(letS (((key value x) (Ealist '((a . 1) (b . 2)))))
  (Rlist (list key value x)))           ;Signals error 8
```

The *expr-list* of a `letS` has the effect of grouping several OSS expressions together. The value of the last form in the *expr-list* is returned as the value of the `letS`. This value may be an OSS value or a non-OSS value.

In addition to placing all of the expressions in the same `letS` binding scope, the grouping imposed by the *expr-list* causes the entire body to become an OSS expression. This can alter the way implicit mapping is applied by including non-OSS functions in the OSS expression.

The restricted nature of OSS variables. There are a number of ways in which the variables bound by `letS` (or `lambdaS` and `defunS`) are more restricted than the ones bound by `let`. For the most part, these restrictions stem from the fact that when the OSS macro package transforms an OSS expression into a loop, it rearranges the expressions extensively. This forces `letS` variable scopes to be supported by variable renaming rather than binding. One result of this is that it is not possible to `declare` (or `proclaim`) a `letS` variable to be special. (Standard Common Lisp does not provide any method for determining whether or not a variable has been proclaimed special. As a result, the OSS macro package is unable to issue an error message when a special `letS` variable is encountered. The Symbolics Common Lisp version of the OSS macro package does issue an error message.)

```
(proclaim '(special z))
(letS ((z (Elist '(1 2 3)))) (Rsum z)) ;erroneous expression
```

Another limitation is that programmers are not allowed to assign values to `letS` variables in the body of a `letS`. (This restriction applies whether or not the variables contain OSS values.) The only time `letS` variables can be given a value is the moment they are bound. (Although assignment could be supported easily enough, the rearrangements introduced by the OSS macro package would make it very confusing for a programmer to figure out exactly what would happen in a given situation. In particular, naively applying implicit mapping to `setq` would lead to peculiar results. In addition, outlawing assignments enhances the functional nature of the OSS macro package.) An error message is issued whenever such an assignment is attempted.

```
(lets ((x (Elist '(1 2 3))))
  (setq x (1+ x))                    ;Signals error 12
  (Rlist x))
```

Another aspect of `letS` variables is that their scope is somewhat limited. In particular, `letS` variables can be referenced in a `letS` or `mapS` which is inside the `letS` which binds them. However, they cannot be referenced in `lambda` or `lambdaS`. (As above, this limitation is imposed in order to avoid confusions due to rearrangements. Further, it is not obvious what it would mean to refer to an OSS variable in a `lambda`. Should some sort of implicit mapping be applied?) No attempt is made to issue error messages in this situation. Rather, the variable reference in question is merely treated as a free variable.

```
(let ((x 4))
  (letS ((x (Elist '(1 2 3))))
    (Rlist (TmapF #'(lambda (y) (+ x y)) x)))) ⇒ (5 6 7)
```

- `letS*` *var-value-pair-list* {*decl*}* `&body` *expr-list* ⇒ *result*

The form `letS*` is exactly the same as `letS` except that the variables are bound sequentially instead of in parallel.

```
(letS* ((x '(1 2 3))
        (y (Elist x))
        (z (Rsum y)))
  (list x (Rmax y) z)) ⇒ ((1 2 3) 3 6)
```

- `prognS &body` *expr-list* ⇒ *result*

As shown below, `prognS` is identical to `letS` except that it cannot contain any variable-value pairs or declarations. It is a degenerate form whose only function is to delineate an OSS expression. This can alter the way implicit mapping is applied by including non-OSS functions in the OSS expression.

```
(prognS . expr-list) ≡ (letS () . expr-list)
```

Complete OSS expressions do not return OSS values. A key point relevant to the discussion above is that syntactically complete OSS expressions are not allowed to return OSS values. This is relevant, because `letS` and `prognS` are often used in such a way that an OSS series gratuitously ends up as the return value. For example, the main intent of the expression below is to print out the elements of the list. However, as written, the expression appears to return an OSS series of the values produced by `prin1`. Because expressions like the one below are relatively common, it was decided not to issue an error message in this situation. Rather, the OSS value is simply discarded and no value is returned.

```
(prognS (prin1 (Elist '(1 2)))) ⇒
;;The output "12" is printed.
```

It might be the case that the programmer actually desires to have a physical series returned in the example above. This can be done by using a reducer such as `Rlist` or `Rvector` as shown below.

```
(prognS (Rlist (prin1 (Elist '(1 2))))) ⇒ (1 2)
;;The output "12" is printed.
```

Preventing complete OSS expressions from returning OSS values does not limit what can be written, because programmers can always return a non-OSS series. This can be a bit cumbersome at times, but it is highly preferable to the large inefficiencies which would be introduced by automatically constructing physical representations for OSS series in situations where the returned values are not used in further computation.

Coercion of Non-Series to Series

If an OSS input of an OSS function is applied to a non-series value, the type conflict is resolved by converting the non-OSS value into a series by inserting `Eoss`. That is to say, a non-OSS value acts the same as an unbounded OSS series of the value.

```
(Ralist (Elist '(a b)) (* 2 3))
  ≡ (Ralist (Elist '(a b)) (Eoss :R (* 2 3))) ⇒ ((a . 6) (b . 6))
```

Using `Eoss` to coerce a non-OSS value to an OSS series has the effect of only evaluating the expression which computes the value once. This has many advantages with regard to efficiency, but may not always be what is desired. Multiple evaluation can be specified by using `TmapF` or `mapS`.

```
(Ralist (Elist '(a b)) (gensym)) ⇒ ((a . #:G004) (b . #:G004))
(Ralist (Elist '(a b)) (TmapF #'gensym)) ⇒ ((a . #:G004) (b . #:G005))
```

Implicit Mapping

Mapping operations can be created by using TmapF. However, in the interest of convenience, two other ways of creating mapping operations are supported. The most prominent of these is implicit mapping. If a non-OSS function appears in an OSS expression and is applied to one or more arguments which are OSS series, the type conflict is resolved by automatically mapping the function over these series.

```
(Rsum (car (Elist '((1) (2)))))
  ≡ (Rsum (TmapF #'car (Elist '((1) (2))))) ⇒ 3
(Rsum (* 2 (Elist '(1 2))))
  ≡ (Rsum (TmapF #'(lambda (x) (* 2 x)) (Elist '(1 2)))) ⇒ 6
```

As shown in the second example, implicit mapping actually applies to entire non-OSS subexpressions rather than merely to individual functions. This promotes efficiency and makes sure that related groups of functions are mapped together. However, it is not always what is desired. For instance, in the first example below, the call on gensym gets mapped in conjunction with the call on `list`. This causes each list to contain a separate gensym variable. It might be the case that the programmer wants to have the same gensym variable in each list. This can be achieved by inserting an Eoss as shown in the second example. (Inserting a Eoss here and there can promote efficiency by avoiding unnecessary recomputation.)

```
(Rlist (list (Elist '(a b)) (gensym)))
  ≡ (Rlist (TmapF #'(lambda (x) (list x (gensym)))
                  (Elist '(a b)))) ⇒ ((a #:G002) (b #:G003))
(Rlist (list (Elist '(a b)) (Eoss :R (gensym))))
  ≡ (Rlist (TmapF #'list
                  (Elist '(a b))
                  (Eoss :R (gensym)))) ⇒ ((a #:G002) (b #:G002))
```

In order to be implicitly mapped, a non-OSS function must appear inside of an OSS expression. For example, the instance of `prin1` in the first example below does not get implicitly mapped, because it is not in an OSS expression. Implicit mapping of the `prin1` can be forced by using progns as shown in the second example above.

```
(prin1 (Elist '(1 2))) ⇒ nil
;;The output "NIL" is printed.
(progns (prin1 (Elist '(1 2)))) ⇒
;;The output "12" is printed.
```

(The result of the first example above is that `NIL` gets printed. This happens because `(Elist '(1 2 3))` is a syntactically complete OSS expression and is therefore not allowed to return a series. It returns no values instead. The function `prin1` demands a value anyway, and gets `nil`.)

Another aspect of implicit mapping is that a non-OSS function will not be mapped unless it is applied to a series. This is usually, but not always, what is desired. Consider the first expression below. The instance of `prin1` is mapped over `x`. However, the instance

of `princ` is not applied to a series and is therefore not mapped. If the programmer intends to print a dash after each number, he has to do something in order to get the `princ` to be mapped. This could be done using `TmapF` or `mapS`. However, the best thing to do is to group the two printing statements into a single subexpression as shown in either of the last two examples below. This grouping shows the relationship between the printing operations and causes them to be mapped together.

```
(letS ((x (Elist '(1 2 3))))
  (prin1 x)
  (princ "-")) ⇒ "-"
;;The output "123-" is printed.
(letS ((x (Elist '(1 2 3))))
  (progn (prin1 x) (princ "-"))) ⇒
;;The output "1-2-3-" is printed.
(letS ((x (Elist '(1 2 3))))
  (format T "~A-" x)) ⇒
;;The output "1-2-3-" is printed
```

Ugly details. Implicit mapping is easy to understand when applied in simple situations such as the ones above. However, it can be applied to any Lisp form. Things become somewhat more complicated when control constructs (e.g., `if`) and binding constructs (e.g., `let`) are encountered. The example below shows the implicit mapping of an `if`. This creates a `lambda` expression containing a conditional which is mapped over a series. A key thing to notice in this example is that implicit mapping of `if` is very different from a use of `Tselect`. In particular, the mapped `if` returns a value corresponding to every input, while the `Tselect` does not.

```
(Rlist (if (plusp (Elist '(10 -11 12))) (Eup)))
  ≡ (Rlist (TmapF #'(lambda (x y) (if (plusp x) y))
                  (Elist '(10 -11 12)) (Eup))) ⇒ (0 nil 2)
(Rlist (Tselect (plusp (Elist '(10 -11 12))) (Eup))) ⇒ (0 2)
```

Another aspect of the way conditionals are handled inside of an OSS expression is illustrated below. When an OSS expression is being processed in order to determine what should be implicitly mapped, the expression is broken up into OSS pieces and non-OSS pieces. If the argument of a conditional is an OSS expression, this argument will end up in a separate piece from the conditional itself. One result of this is that the argument will always be evaluated and the conditional will therefore lose its power to control when the argument should be evaluated. This effect will happen even if, as in the example below, the conditional does not have to be mapped. The three examples below all produce the same value, but the first two always evaluate `(Rlist (abs (Elist x)))` while the last may not.

```
(prognS (if (Ror (minusp (Elist x)))
            (Rlist (abs (Elist x)))
            x))
  ≡ (prognS (funcall #'(lambda (y z) (if y z x))
                     (Ror (minusp (Elist x)))
                     (Rlist (abs (Elist x)))))
  ≢ (if (Ror (minusp (Elist x)))
        (Rlist (abs (Elist x)))
        x)
```

The following example shows the implicit mapping of a `let`. (Among other things, this illustrates that such expressions are far from clear. In general it is better to use `letS` as in the second example.)

```
(Rlist (let ((double (* 2 (Elist '(1 2))))) (* double double)))
  ≡ (Rlist (TmapF #'(lambda (x)
                      (let ((double (* 2 x))) (* double double)))
                  (Elist '(1 2)))) ⇒ (4 16)

(letS ((double (* 2 (Elist '(1 2)))))
  (Rlist (* double double))) ⇒ (4 16)
```

A problem with the implicit mapping of a `let` (or other binding forms) is that the implicit mapping transformation potentially moves subexpressions out of the scope of the binding form in question. This can change the meaning of the expression if any of these subexpressions contain an instance of a variable bound by the binding form. For instance, in the example above, the transformation moves the subexpression `(Elist '(1 2))` out of the scope of the `let`. This would cause a problem if this subexpression referred to the variable `double`.

In recognition of this problem, a warning message is issued whenever implicit mapping of a binding form causes a variable reference to move out of a form that binds it. Whenever it occurs, this problem can be alleviated by using `letS` as shown above.

A final complexity involves forms like `return`, `return-from`, `throw`, etc. These forms are implicitly mapped like any other non-OSS form. When they get evaluated, they will cause an exit. However, the loop produced by the OSS macro does not contain a boundary which is recognized by any of these forms (e.g., it does not create a `prog` or `catch`). As a result, such a boundary must be defined which will serve as the reference point. Needless to say, the final results of the OSS expression will not be computed if the expression is exited in this way.

Nested loops. Implicit mapping is applied when non-OSS functions receive OSS values. However, implicit mapping is not applied when OSS functions receive OSS values, even if these values are passed to non-OSS inputs. As illustrated below, whenever this situation occurs, an error message is issued.

```
(Elist (Elist '((1 2) (3 4))))                    ;Signals error 14
```

There are situations corresponding to nested loops where it would be reasonable to implicitly map subexpressions containing OSS functions. For example, one might write the following expression in order to copy a list of lists.

```
(Rlist (Rlist (Elist (Elist '((1 2) (3 4))))))  ;Signals error 14
(Rlist (TmapF #'(lambda (x) (Rlist (Elist x)))
              (Elist '((1 2) (3 4))))) ⇒ ((1 2) (3 4))
```

Nevertheless, expressions like the first one above are forbidden. This is done for two reasons. First, in more complex situations OSS expressions corresponding to nested loops become so confusing that such expressions are very hard to understand. As a result, they are not very useful. Second, experience suggests that a large proportion of

situations where mapping of OSS functions might be done arise from programming errors rather than an intention to have a nested loop. Outlawing these expressions makes it possible to find these errors more quickly.

(The following example shows that there is no problem with having one loop computation following another. There are no type conflicts in this situation and no implicit mapping is required.)

```
(Rsum (Evector (Rvector (Elist '(1 2))))) ⇒ 3
```

Needless to say, it would be unreasonable if there were no way to write OSS expressions corresponding to nested loops. First of all, this can always be done using `TmapF` as shown above. However, this can be rather cumbersome. To alleviate this difficulty, an additional form (`mapS`) is introduced which facilitates the expression of nested computations.

- `mapS &body` *expr-list* ⇒ *items*

The *expr-list* consists of one or more expressions. These expressions are treated as the body of a function and mapped over any free OSS variables which appear in them. That is to say, the first element of the output is computed by evaluating the expressions in an environment where each OSS variable is bound to the first element of the corresponding series. The second element of the output is computed by evaluating the expressions in an environment where each OSS variable is bound to the second element of the corresponding series, etc. The way `mapS` could be used to copy a list-of-lists is shown below. A `letS` has to be used, because `mapS` requires that the series being mapped over must be held in a variable.

```
(letS ((z (Elist '((1 2) (3 4)))))
  (Rlist (mapS (Rlist (Elist z)))))
  ≡ (letS ((z (Elist '((1 2) (3 4)))))
      (Rlist (TmapF #'(lambda (x)
                        (Rlist (Elist x))) z))) ⇒ ((1 2) (3 4))

(Rlist
  (mapS
    (Rlist (Elist (Elist '((1 2) (3 4)))))) ;Signals error 14
```

Implicit mapping is very valuable. From the above, it can be seen that although implicit mapping is simple in simple situations, there are a number of situations where it becomes quite complex. There is no question that these complexities dilute the value of implicit mapping. Nevertheless, experience suggests that implicit mapping is so valuable that, warts and all, it is perhaps the most useful single feature of OSS expressions.

Literal Series Functions

Just as it is very convenient to be able to specify a literal non-OSS function using `lambda`, it is sometimes convenient to be able to specify a literal OSS function.

- `lambdaS` *var-list* {*decl*}* `&body` *expr-list*

The form `lambdaS` is analogous to `lambda` except that some of the arguments can have OSS series passed to them and the return value can be an OSS series. The `var-list` is

simpler than the `lambda` lists which are supported by `lambda`. In particular, the `var-list` must consist solely of variable names. It cannot contain any of the `lambda` list keywords such as `&optional` and `&rest`. As in a `letS`, the variables in the *var-list* cannot be assigned to in the `expr-list` or referenced inside of a nested `lambda` or `lambdaS`.

As in a `lambda`, the body can begin with one or more declarations. All of the arguments which are to receive OSS values have to be declared inside the `lambdaS` using the declaration `type oss` (see below). All of the other arguments are assumed to correspond to non-OSS values. Just as in a `letS`, the declarations may contain other kinds of declarations besides `type oss` declarations. However, the variables in the `var-list` cannot be declared (or proclaimed) to be special.

The `expr-list` is a list of expressions which are grouped together into an OSS expression as in a `letS` or `prognS`. The value of the function specified by a `lambdaS` is the value of the last form in the *expr-list*. This value may or may not be an OSS series.

In many ways, `lambdaS` bears the same relationship to `letS` that `lambda` bears to `let`. However, there is one key difference. The *expr-list* in a `lambdaS` cannot refer to any free variables which are bound by a `letS`, `defunS`, or another `lambdaS`. Each `lambdaS` is processed in complete isolation from the OSS expression which surrounds it. The only values which can enter or leave a `lambdaS` are specified by the `var-list` and non-OSS variables which are bound outside of the entire containing OSS expression.

Another key feature of `lambdaS` is that the only place where it can validly appear is as the quoted first argument of `funcallS` (see below), or as an argument to a macro which will eventually expand in such a way that the `lambdaS` will end up as the quoted first argument of a `funcallS`.

The following example illustrates the use of `lambdaS`. It shows an anonymous OSS function identical to `Rsum`.

```
(funcallS #'(lambdaS (x)
                (declare (type oss x))
              (ReduceF 0 #'+ x))
          (Elist '(1 2 3))) ⇒ 6
```

- `type oss &rest` *variable-list*

  This type declaration can only be used inside of a `declare` inside of a `lambdaS` or a `defunS`. It specifies that the variables carry OSS values.

- `funcallS` *function* `&rest` *expr-list* ⇒ *result*

  This is analogous to `funcall` except that *function* can be an OSS function. In particular, it can be the quoted name of a series function, a quoted `lambdaS`, or a macro call which expands into either of the above. It is also possible for *function* to be a non-OSS function, in which case `funcallS` is identical to `TmapF`. If *function* is an expression which evaluates to a function (as opposed to a literal function), then it is assumed to be a non-OSS function.

```
(funcallS #'Elist '(1 2)) ≡ (Elist '(1 2)) ⇒ [1 2]
(funcallS #'(lambdaS (y) (declare (type oss y)) (* 2 y))
          (Elist '(1 2))) ⇒ [2 4]
(funcallS #'car [(1) (2)]) ⇒ [1 2]
(funcallS #'car '(1 2)) ⇒ [1 1 1 1 ...]
```

The number of expressions in *expr-list* must be exactly the same as the number of arguments expected by *function*. If not, an error message is issued. In addition, the types of values (either OSS series or not) returned by the expressions should be the same as the types which are expected by *function*. If not, coercion of non-series to series will be applied if possible in order to resolve the conflict.

Defining Series Functions

An important aspect of the OSS macro package is that it makes it easy for programmers to define new OSS functions. Straightforward OSS functions can be defined using the facilities outlined below. More complex OSS functions can be defined using the subprimitive facilities described in [6].

- `defunS` *name lambda-list* {*doc*} {*decl*}* `&body` *expr-list*

This is analogous to `defun`, but for OSS functions. At a simple level, `defunS` is just syntactic sugar which defines a macro that creates a `funcallS` of a `lambdaS`. The *lambda-list*, declarations, and expression list are restricted in exactly the same way as in a `lambdaS` except that the standard `lambda` list keywords `&optional` and `&key` are allowed in the *lambda-list*.

```
(defunS Rlast (items &optional (default nil))
    "Returns the last element of an OSS series"
    (declare (type oss items))
  (ReduceF default #'(lambda (state x) x) items))
  ≡ (defmacro Rlast (items &optional (default 'nil))
      "Returns the last element of an OSS series"
     `(funcallS #'(lambdaS (items default)
                    (declare (type oss items))
                  (ReduceF default #'(lambda (state x) x) items))
                ,items ,default))
```

However, at a deeper level, there is a key additional aspect to `defunS`. Preprocessing and checking of the resulting `lambdaS` is performed when the `defunS` is evaluated (or compiled), rather than when the resulting OSS function is used. This saves time when the function is used. More importantly, it leads to better error messages because error messages can be issued when the `defunS` is initially encountered, rather than when the OSS function defined is used.

Although the `lambda` list keywords `&optional` and `&key` are supported by `defunS`, it should be realized that they are supported in the way they are supported by macros, not the way they are supported by functions. In particular, when keywords are used in a call on the OSS function being defined, they have to be literal keywords rather than computed by an expression. In addition, initialization forms cannot refer to the run-time values of other arguments, because these are not available at macro-expansion-time. They are also not allowed to refer to the macro-expansion-time values of the other arguments. They must stand by themselves when computing a value. A quote is inserted so that this value will be computed at run-time rather than at macro-expansion-time. (In the example above, `(default nil)` becomes `(default 'nil)`.)

It may seem unduly restrictive that `defunS` does not support all of the standard keywords in *lambda-list*. However, this is not that much of a problem because `defmacro` can be used directly in situations where these capabilities are desired. For example, `Tconcatenate` is defined in terms of a more primitive OSS function `Tconcatenate2` as follows.

```
(defmacro Tconcatenate (Oitems1 Oitems2 &rest more-Oitems)
  (if (null more-Oitems)
      `(Tconcatenate2 ,Oitems1 ,Oitems2)
      `(Tconcatenate2 ,Oitems1 (Tconcatenate ,Oitems2 .,more-Oitems))))
```

Using `defmacro` directly also makes it possible to define new higher-order OSS functions. For example, an OSS function analogous to `substitute-if` could be defined as follows. (The `Eoss` ensures that `newitem` will only be evaluated once.)

```
(defmacro Osubstitute-if (newitem test items)
  (let ((var (gensym)))
    `(letS ((,var ,items))
       (if (funcall ,test ,var) (Eoss :R ,newitem) ,var))))
(Osubstitute-if 3 #'minusp [1 -1 2 -3]) ⇒ [1 3 2 3]
```

Multiple Values

The OSS macro package supports multiple values in a number of contexts. As discussed above, `letS` can be used to bind variables to multiple values returned by an OSS function. Faculties are also provided for defining OSS functions which return multiple values. The support for multiple values is complicated by the fact that the OSS macro package implements all communication of values by using variables. As a result, it is not possible to support the standard Common Lisp feature that multiple values can coexist with single values without the programmer having to pay much attention to what is going on. When using OSS expressions, the programmer has to be explicit about how many values are being passed around.

- `valS &rest` *expr-list* ⇒ `&rest` *multiple-value-result*

  This is analogous to `values` except that it can operate on OSS values. It takes in the values returned by n different expressions and returns them as n multiple values. It enforces the restriction that the values must either all be OSS values or all be non-OSS values. The following example shows how a simple version of `Eplist` could be defined.

```
(defunS simple-Eplist (place)
  (letS ((plist (EnumerateF place #'cddr #'null)))
    (valS (car plist) (cadr plist))))
```

It is possible to use `values` in an OSS expression. However, the results will be very different from the results obtained from using `valS`. The `values` will be implicitly mapped like any other non-OSS form. The value ultimately returned will be the single value returned by `TmapF`.

```
(prognS (valS (Elist '(1 2)) (Elist '(3 4)))) ⇒ [1 2] [3 4]

(prognS (values (Elist '(1 2)) (Elist '(3 4))))
  ≡ (prognS (TmapF #'(lambda (x y) (values x y))
                   (Elist '(1 2)) (Elist '(3 4)))) ⇒ [1 2]
```

- `pass-valS` *n expr* ⇒ `&rest` *multiple-value-result*

This function is used essentially as a declaration. It tells the OSS macro package that the form *expr* returns *n* multiple values which the programmer wishes to have preserved in the context of the OSS expression. (This is needed, because Common Lisp does not provide any compile-time way to determine the number of arguments that a function will return.) The first example below enumerates a list of symbols and returns a list of the internal symbols, if any, which correspond to them. The second example defines a two valued OSS function which locates symbols.

```
(letS* ((names (Elist '(zots Elist zorch)))
        ((symbols statuses) (pass-valS 2 (find-symbol (string names))))
        (internal-symbols (Tselect (eq statuses :internal) symbols)))
  (Rlist internal-symbols)) ⇒ (zots zorch)

(defunS find-symbols (names)
    (declare (type oss names))
  (pass-valS 2 (find-symbol (string names))))

(find-symbols [zots Elist zorch])
  ⇒ [zots Elist zorch] [:internal :inherited :internal]
```

The form `pass-valS` never has to be used in conjunction with an OSS function, because the OSS macro package knows how many values every OSS function returns. Similarly, `pass-valS` never has to be used when multiple values are being bound by `letS`, because the syntax of the `letS` indicates how many values are returned. (As a result, the `pass-valS` in the first example above is not necessary.) However, in situations such as the second example above, `pass-valS` must be used.

Alteration of Values

The transformations introduced by the OSS macro package are inherently antagonistic to the transformations introduced by the macro `setf`. In particular, OSS function calls cannot be used as the destination of a `setf`. In order to get around this problem, the OSS macro package supports a separate construct which is in fact more powerful than `setf`.

- `alterS` *destinations items* ⇒ *items*

This form takes in a series of destinations and a series of items and stores the items in the destinations. It returns the series of items. Like `setf`, `alterS` cannot be applied to a destination unless there is an associated definition for what should be done (see the discussion of `alterableS` in [6]). The outputs of the predefined functions `Elist`, `Ealist`, `Eplist`, `Efringe`, `Evector`, and `Esequence` are alterable. The effects of this alteration are illustrated in conjunction with the descriptions of these functions. For example, the following sets all of the elements in a list to `nil`.

```
(let ((list '((a . 1) (b . 2) (c . 3))))
  (alterS (Elist list) nil)
  list) ⇒ (nil nil nil)
```

As a related example, consider the following. Although `setf` cannot be applied to an OSS function, it can be applied to a non-OSS function in an OSS expression. In the example below, `setf` is used to set the cdr of each element of a list to `nil`.

```
(let ((list '((a . 1) (b . 2) (c . 3))))
  (prognS (setf (cdr (Elist list)) nil))
  list) ⇒ ((a) (b) (c))
```

A key feature of `alterS` is that (in contrast to `setf`) a structure can be altered by applying `alterS` to a variable which contains enumerated elements of the structure. This is useful because the old value in a structure can be used to decide what new value should be put in the structure. (When `alterS` is applied to such a variable it modifies the structure being enumerated but does not change the value of the variable.)

```
(letS* ((v '#(1 2 3))
        (x (Evector v)))
  (alterS x (* x x))
  (valS (Rlist x) v)) ⇒ (1 2 3) #(1 4 9)
```

Another interesting aspect of `alterS` is that it can be applied to the outputs of a number of transducers. This is possible whenever a transducer passes through unchanged a series of values taken from an input which is itself alterable. This can happen with the transducers `Tuntil`, `TuntilF`, `Tcotruncate`, `Tremove-duplicates`, `Tsubseries`, `Tselect`, `TselectF`, `Tsplit`, and `TsplitF`. For example, the following takes the absolute value of the elements of a vector.

```
(letS* ((v '#(1 -2 3))
        (x (TselectF #'minusp (Evector v))))
  (alterS x (- x))
  v) ⇒ #(1 2 3)
```

Debugging

The OSS macro package supports a number of features which are intended to facilitate debugging. One example of this is the fact that the macro package tries to use the variable names which are bound by a `letS` in the code produced. Since the macro package is forced to use variable renaming in order to implement variable scoping, it cannot guarantee that these variable names will be used. However, there is a high probability that they will. If a break occurs in the middle of an OSS expression, these variables can be inspected in order to determine what is going on. If a `letS` variable holds an OSS series, then the variable will contain the current element of the series. For example, the OSS expression below is transformed into the loop shown. (For a discussion of how this transformation is performed see [6].)

```
(letS* ((v (get-vector user))
        (x (Evector v)))
  (Rsum x))

(let (#:index-9 #:last-8 #:sum-2 x v)
  (setq v (get-vector user))
  (tagbody (setq #:index-9 -1)
           (setq #:last-8 (length v))
           (setq #:sum-2 0)
     #:L-1 (incf #:index-9)
           (if (not (< #:index-9 #:last-8)) (go oss:END))
           (setq x (aref v #:index-9))
           (setq #:sum-2 (+ #:sum-2 x))
           (go #:L-1)
   oss:END)
  #:sum-2)
```

- showS *thing* &optional (*format* "~%~S") (*stream* *standard-output*) ⇒ *thing*

This function is convenient for printing out debugging information while an OSS expression is being evaluated. It can be wrapped around any expression no matter whether it produces an OSS value or a non-OSS value without disturbing the containing expression. The function prints out the value and then returns it. If the value is a non-OSS thing, it will be printed out once at the time it is created. If it is an OSS series thing, it will be printed out an element at a time. The *format* can be used to print a tag in order to identify the value being shown.

```
(showS format stream)
  ≡ (let ((x thing)) (format stream format x) x)

(letS ((x (Elist '(1 2 3))))
  (Rsum (showS x "Item: ~A, "))) ⇒ 6
;;The output "Item: 1, Item: 2, Item: 3, " is printed.
```

- *permit-non-terminating-oss-expressions*

On the theory that non-terminating loops are seldom desired, the OSS macro package checks each loop constructed to see if it can terminate. If this control variable is nil (which is the default), then a warning message is issued for each loop which the OSS macro package thinks has no possibility of terminating. This is useful in the first example below, but not in the second. The form compiler-let can be used to bind this control variable to T around such an expression.

```
(Rlist 4)                       ;Signals warning 15

(block bar                      ;Signals warning 15
  (letS ((x (Eup :by 10)))
    (if (> x 15) (return-from bar x)))) ⇒ 20

(compiler-let ((*permit-non-terminating-oss-expressions* T))
  (block bar
    (letS ((x (Eup :by 10)))
      (if (> x 15) (return-from bar x))))) ⇒ 20
```

- `*last-oss-loop*`

  This variable contains the loop most recently produced by the OSS macro package. After evaluating (or macro-expanding) an OSS expression, this variable can be inspected in order to see the code which was produced.

- `*last-oss-error*`

  This variable contains the most recently printed warning or error message produced by the OSS macro package. The information in this variable can be useful for tracking down errors.

Side-Effects

The OSS macro package works by converting each OSS expression into a loop. This allows the expressions to be evaluated very efficiently, but radically changes the order in which computations are performed. In addition, off-line ports are supported by code motion. Given all of these changes, it is not surprising that OSS expressions are primarily intended to be used in situations where there are no side-effects. Due to the change in computation order, it can be hard to figure out what the result of a side-effect will be.

Nevertheless, since side-effects (particularly in the form of input and output) are an inevitable part of programming, several steps are taken in order to make the behavior of OSS expressions containing side-effect operations as easy to understand as possible. First, when implicit mapping is applied, it is applied to as large a subexpression as possible. This makes it straightforward to understand the interaction of the side-effects within a single mapped subexpression. Several examples of this are given in the section above which discusses implicit mapping.

Second, wherever possible, the OSS macro package leaves the order of evaluation of the OSS functions in an expression unchanged. Each function is evaluated incrementally an element at a time, but on each cycle, the processing follows the syntactic ordering of the functions in the expression.

The one place where order changes are required is when handling off-line ports. However, things are simplified here by ensuring that the evaluation order implied by the order of the inputs of an off-line function is preserved.

Third, when determining whether or not each termination point is connected to every output in each on-line subexpression, functions whose outputs are not used for anything are considered to be outputs of the subexpression. The reasoning behind this is that if the outputs are not used for anything, then the function must be being used for side-effect and probably matters that the function get evaluated the full number of times it should be. For example, consider the expressions below. The first expression prints out the numbers in a list and returns the first negative number. The second expression signals a warning and the enumeration of the list is duplicated so that the `princ` will be applied to all of the elements of the list.

```
(letS* ((x (Elist '(1 2 3 -4 5))))
  (princ x)
  (Rfirst-passive (TselectF #'minusp x))) ⇒ -4
;;The output "123-45" printed.
(letS* ((x (Elist '(1 2 3 -4 5))))            ;Signals warning 18
  (princ x)
  (Rfirst (TselectF #'minusp x))) ⇒ -4
;;The output "123-45" printed.
```

3. Bibliography

[1] A. Aho, J. Hopcraft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading MA, 1974.

[2] G. Burke and D. Moon, *Loop Iteration Macro*, MIT/LCS/TM-169, July 1980.

[3] R. Polivka and S. Pakin, *APL: The Language and Its Usage*, Prentice-Hall, Englewood Cliffs NJ, 1975.

[4] G. Steele Jr., *Common Lisp: the Language*, Digital Press, Maynard MA, 1984.

[5] R. Waters, "A Method for Analyzing Loop Programs", *IEEE Trans. on Software Engineering*, 5(3):237–247, May 1979.

[6] R. Waters, *Synchronizable Series Expressions: Part II: Overview of the Theory and Implementation*, MIT/AIM-959, November 1987

[7] *Lisp Machine Documentation for Genera 7.0*, Symbolics, Cambridge MA, 1986.

4. Warning and Error Messages

In order to facilitate the debugging of OSS expressions, this section discusses the various warning and error messages which can be issued by the OSS macro package while processing the functions described in this document. Error messages describe problems in OSS expressions which make it impossible to process the expression correctly. Warning messages identify less serious situations which are worthy of programmer scrutiny, but which do not prevent the expression from being processed in a way which is, at least probably, correct.

Warning and error messages are both printed out in the following format. Error messages (as opposed to warnings) can be identified by the fact that the word "Error" precedes the message number. (The format is shown as it appears on the Symbolics Lisp machine and may differ in minor ways in other systems.)

```
Warning: {Error} message-number in OSS expression:
containing OSS expression
detailed message
```

For example, the following error message might be printed.

```
Warning: Error 1.1 in OSS expression:
(LETS ((X (ELIST NUMBER-LIST))
       (Y (EUP (CAR HEADER) :TO 4 :LENGTH 5)))
  (RLIST (LIST Y X)))
Too many keywords specified in a call on Eup:
(EUP (CAR HEADER) :TO 4 :LENGTH 5)
```

The first line of each message specifies the number of the warning or error. This number is useful for looking up further information in the documentation below. The next part of the message shows the complete OSS expression which contains the problem. This makes it easier to locate the problem in a program. The remainder of the message describes the particular problem in detail. (The variable `*last-oss-error*` contains a list of the information which was used to print out the most recent warning or error message.)

The OSS macro package reports problems using `warn` so that processing of other parts of a program can continue, potentially finding other problems. However, each time an OSS error (as opposed to a warning) is detected, the OSS macro package skips over the rest of the OSS expression without performing any additional checks. Therefore, even if there are several OSS errors in an OSS expression, only one OSS error will be reported. When an OSS error is found, a dummy value is inserted in place of the erroneous OSS expression. As a result, it is virtually impossible for the containing program to run correctly.

The documentation below describes each of the messages which the OSS macro package can produce. Each description begins with a header line containing a schematic rendition of the message. Italics is used to indicate pieces of specific information which are inserted in the message. The number of the warning or error is shown in the left margin at the beginning of the header. For ease of reference, the messages are described in numerical order.

Local errors concerning single OSS functions. The following error messages report errors which are local in that they stem purely from the improper use of a single OSS function. These errors cover only a few special situations. Many (if not most) local errors are reported directly by the standard Common Lisp processor rather than by the OSS macro package. For example, if an OSS function is used with the wrong number of arguments, an error message is issued by the standard macro expander.

1.1 Error: `Too many keywords specified in call on Eup:` *call*
1.2 Error: `Too many keywords specified in call on Edown:` *call*
1.3 Error: `Too many keywords specified in call on Tlatch:` *call*

Each of these errors specifies that incompatible keywords have been provided for the indicated function. The entire function call is printed out as shown above.

2 Error: `Invalid enumerator arg to TconcatenateF:` *enumerator*

This error is issued if the enumerator argument to `TconcatenateF` fails to be an enumerator—i.e., fails to be an OSS function that has no OSS inputs, at least one OSS output, and which can terminate.

3 Error: `Unsupported &-keyword` *keyword* `in defunS arglist.`

This error is issued if an `&-keyword` other than `&optional` or `&key` appears in the argument list of `defunS`. Other keywords have to be supported by using `defmacro` directly. (See the discussion of `defunS`.)

4 Error: `AlterS applied to an unalterable form:` *call*

This error is issued if `alterS` is applied to a value which is not alterable. Values are alterable only if they come directly from an enumerator which has an alterable value, or come indirectly from such an enumerator via one or more transducers which allow alterability to pass through.

5 Error: `Malformed lambdaS argument` *arg*.

This error message is issued if an argument of a `lambdaS` fails to be a valid variable. In particular, it is issued if the argument, is not a symbol, is `T` or `nil`, is a symbol in the keyword package, or is an `&-keyword`. (It is also erroneous for such a variable to be declared special. However, this error is only reported on the Symbolics Lisp Machine.)

6 Error: `LambdaS used in inappropriate context:` *call*

This error message is issued if a `lambdaS` ends up (after macro expansion of the surrounding code) being used in any context other than as the quoted first argument of a `funcallS`.

7 Error: `Wrong number of args to funcallS:` *call*

This error message is issued if a use of `funcallS` does not contain a number of arguments which is compatible with the number of arguments expected by the OSS functional argument.

8 Error: `Only` *n* `return values present where` *m* `expected:` *call*

This error message is issued if an OSS function is used in a situation where it is expected to return more values than it actually does—for example, if a `letS` tries to bind two values from an OSS function which only returns one, or `pass-valS` tries to obtain two values from an OSS function which only returns one. (Non-OSS functions return extra values of `nil` if they are requested to produce more values than they actually do.)

Warnings and errors concerning OSS variables. The following warnings and errors concern the creation and use of `letS` and `lambdaS` variables. Like the errors above, they are quite local in nature and relatively easy to fix.

9 Error: `Malformed letS{*} binding pair` *pair*.

This error message is issued if a `letS` or `letS*` binding pair fails to be either a list of a valid variable and a value, or a list of a list of valid variables and a value. The criterion for what makes a variable valid is the same as the one used in Error 5, except that a binding pair can contain `nil` instead of a variable.

10 Warning: `The variable(s)` *vars* `declared TYPE OSS in a letS{*}.`

This warning message is issued if one or more variables in a `letS` are explicitly declared to be of type `oss`. The explicit declarations are ignored.

11 Warning: `The letS{*} variable` *variable* `is unused in:` *call*

This warning message is issued if a variable in a `letS` is never referenced in the body of the `letS`. Note that these variables cannot be referenced inside a nested `lambda` or `lambdaS`.

12 Error: `The letS{*} variable` *var* `setqed.`

This error message is issued if a `letS` variable (either OSS or non-OSS) is assigned to in the body of a `letS`. It is also issued if any of the variables bound by a `lambdaS` or `defunS` are assigned to.

Non-local warnings and errors concerning complete OSS expressions. The following warnings and errors concern non-local problems in OSS expressions. The first two are discussed in further detail in the section on implicit mapping.

13 Warning: `Decomposition moves:` *code* `out of a binding scope:` *surround*

This warning is issued if the processing preparatory to implicit mapping causes a subexpression to be moved out of the binding scope for one of the variables in it. The problem can be fixed by using `letS` to create the binding scope, or by moving the binding form so that it surrounds the entire OSS expression. (The testing for this problem is somewhat approximate in nature. It can miss some erroneous situations and can complain in some situations where there is no problem. Due to this latter difficulty, the OSS macro package merely issues a warning message rather than issuing an error message.)

14 Error: `OSS value carried to non-OSS input by data flow from:` *call* `to:` *call*

As illustrated below, this error is issued whenever data flow connects an OSS output to a non-OSS input of an OSS function as in the example below. (If the expression in question is intended to contain a nested loop, the error can be fixed by wrapping the nested portion in a mapS.)

```
Warning: Error 14 in OSS expression:
(Rlist (Rlist (Elist (Elist '((1 2) (3 4))))))
OSS value carried to non-OSS input by data flow from:
(Elist '((1 2) (3 4)))
to:
(Elist (Elist '((1 2) (3 4))))
```

The error message prints out two pieces of code in order to indicate the source and destination of the data flow in question. The outermost part of the first piece of code shows the function which creates the value in question. The outermost function in the second piece of code shows the function which receives the value. (Entire subexpressions are printed in order to make it easier to locate the functions in question within the OSS expression as a whole.) If nesting of expressions is used to implement the data flow, then the first piece of code will be nested in the second one.

15 Warning: `Non-terminating OSS expression:` *expr*

This warning message is issued whenever a complete OSS expression appears incapable of terminating. The expression in question is printed. It may well be only a subexpression of the OSS expression being processed. A warning message is issued instead of an error message, because the expression may in fact be capable of terminating or the expression might not be intended to terminate. (This warning message can be turned off by using the variable `*permit-non-terminating-oss-expressions*`.)

Warnings concerning the violation of restrictions. The following warnings are issued when an OSS expression violates one of the isolation restrictions or the requirement that within each on-line subexpression, there must be a data flow path from each termination point to each output. In each case, the violation is automatically fixed by the macro package. However, in order to achieve high efficiency, the user should fix the violation explicitly rather than relying on the automatic fix.

16 Warning: `Non-isolated non-oss data flow from:` *call* `to:` *call*

This warning is issued if an OSS expression violates the non-OSS data flow isolation restriction. As shown below, the message prints out two pieces of code which indicate the data flow in question.

```
Warning: 16 in OSS expression:
(LETS* ((NUMS (EVECTOR '#(3 2 8)))
        (TOTAL (REDUCEF 0 #'+ NUMS)))
  (RVECTOR (/ NUMS TOTAL)))
Non-isolated non-OSS data flow from:
(REDUCEF 0 #'+ NUMS)
to:
(/ NUMS TOTAL)
```

As discussed on page 10, the OSS macro package automatically fixes the isolation restriction violation by duplicating subexpressions until the data flow in question becomes isolated. (In the example above, the vector enumeration gets copied.) However, the macro package is not guaranteed to minimize the amount of code copied. In addition, it is sometimes possible for a programmer to fix an expression much more efficiently without using any code copying. As a result, it is advisable for programmers to fix these violations explicitly, rather than relying on the automatic fixes provided by the OSS macro package.

17.1 Warning: `Non-isolated oss input at the end of the data flow from:` *call* `to:` *call*

17.2 Warning: `Non-isolated oss output at the start of the data flow from:` *call* `to:` *call*

One of these warnings is issued if an OSS expression violates the off-line port isolation restriction. The warning message prints out two pieces of code which indicate a data flow which ends (or starts) on the port in question. Code copying is automatically applied in order to fix the violation. As discussed on page 10, it is worthwhile to try and think of a more efficient way to fix the violation. As with Warning 16, even if code copying is the only thing which can be done, it is better for the programmer to do this explicitly.

18 Warning: `No data flow path from the termination point:` *call* `to the output:` *call*

This warning is issued if a termination point in an on-line subexpression of an OSS expression is not connected by data flow to one of the outputs. Code copying is automatically applied in order to fix the violation. (However, the OSS macro package has a tendency to copy a good deal more code than necessary.) As discussed on page 12, the violation can often be fixed much more efficiently by using non-early-terminating OSS functions instead of early-terminating functions or by using `Tcotruncate` to indicate relationships between inputs.

Errors concerning implementation limitations. These errors reflect limitations of the way the OSS macro package is implemented rather than anything fundamental about OSS expressions.

19 Error: `LambdaS body too complex to merge into a single unit:` *forms*

In general, the OSS macro package is capable of combining together any kind of permissible OSS expression. In particular, there is never a problem as long as the expression as a whole does not have any OSS inputs or OSS outputs. However, in the body of a `lambdaS`, it is possible to write OSS expressions which have both OSS inputs and OSS outputs. If such an expression has a data flow path from an OSS input to an OSS output which contains a non-OSS data flow arc, then this error message is issued. For example, the error would be issued in the situation below.

```
(funcallS #'(lambdaS (items)                          ;Signals error 19
              (declare (type oss items))
            (Elist (Rlist items)))
          ...)
```

An error message is issued in the situation above, because the situation is unlikely to occur and there is no way to support the situation without resorting to very peculiar

code. In particular, the input `items` in the example above would have to be converted into an off-line input.

20 Error: `The form` *function* `not allowed in OSS expressions.`

In general, the OSS macro package has a sufficient understanding of special forms to handle them correctly when they appear in an OSS expression. However, it does not handle the forms `compiler-let`, `flet`, `labels`, or `macrolet`. The forms `compiler-let` and `macrolet` would not be that hard to handle, however it does not seem worth the effort. The forms `flet` and `labels` would be hard to handle, because the OSS macro package does not preserve binding scopes and therefore does not have any obvious place to put them in the code it produces. All four forms can be used by simply wrapping them around entire OSS expressions rather than putting them in the expressions.

21–27 Documentation for these errors appears in [6].

5. Index of Functions

This section is an index and concise summary of the functions, variables, and special forms described in this document. Each entry shows the inputs and outputs of the function, the page where documentation can be found, and a one line description.

The names of OSS functions often start with one of the following prefix letters.

`E` Enumerator.
`T` Transducer.
`R` Reducer.

Occasionally, a name will end with one of the following suffix letters.

`S` Special form.
`F` Function that takes functional arguments.

In addition, the argument and result names indicate data type restrictions (e.g., *number* indicates that an argument must be a number, *item* indicates that there is no type restriction). Plural names are used iff the value in question is an OSS series (e.g., *numbers* indicates an OSS series of numbers; *items* indicates an OSS series of unrestricted values). The name of a series input or output begins with "`O`" iff it is off-line.